INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




GraphQL queries: questões de segurança e performance

LUÍS BRUNO VIEIRA MONTEIRO
junho de 2022

POLITÉCNICO
DO PORTO

GraphQL queries: security and performance concerns

Luís Monteiro

A dissertation submitted in partial fulfillment of the requirements for the degree of Informatics Engineering, Specialisation Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, June 29, 2022

Abstract

As web applications continue to grow, the need for these applications to adapt to the needs of the client is more important. Applications like mobile and web require efficient responses. The search for alternatives to the very popular REST APIs gains more interest.

One popular approach is GraphQL. GraphQL addresses both the flexibility and performance concerns that can occur in RESTFul APIs by allowing users to request the data as they need it. With this flexibility, security and performance concerns become more and more relevant.

This work aims to explore how can these concerns be answered by analysing queries and deciding if they are safe to be answered or not. Always having in mind that the performance should not be affected.

One of the conclusions found is that the implementation of a simpler strategy is fundamental to any GrahQL API since it provides a minimum level of security without compromising performance. However, the use of a more complex approach offers better results, without having any major disadvantage.

Keywords: GraphQL, Queries, Expensive, Cost, Analysis, Security

Resumo

Com o constante crescimento de aplicações web, a necessidade destas se adaptarem à necessidade dos seus cliente torna-se mais importante. Aplicações mobile e web necessitam de respostas eficientes. A procura de novas alternativas para as populares REST APIs ganha mais interesse.

Uma solução popular é GraphQL. GraphQL responde diretamente às questões quer de flexibilidade, quer de performance, que podem ocorrer em RESTFul APIs, permitindo aos seus utilizadores pedir as informações conforme necessitem. Com esta flexibilidade, questões relacionadas com a segurança e performance tornam-se mais relevantes.

Este trabalho pretende explorar como é que estas questões podem ser resolvidas através da análise das queries, decidindo se as mesmas são seguras ou não. Tendo sempre em conta que a performance não deve ser afetada.

Uma das conclusões obtidas é de que a implementação de uma estratégia mais simples é fundamental pois oferece um nível básico de segurança sem compremeter a performance. Contudo, o use de estratégias mais complexas oferece melhores resultados, sem ter desvantagens claras.

Table of Contents

List of Figures

List of Tables

List of Source code extracts

List of Acronyms

AHP	Analytic Hierarchy Process.
API	Application Programming Interface.
DoS	Denial of Service.
GraphQL	Graph Query Language.
IP	Internet Protocol.
ISEP	Instituto Superior de Engenharia do Porto.
NCD	New Concept Development.
OOP	Object Oriented Programming.
REST	Representational State Transfer.
SSD	Software Sequence Diagram.
UI	User Interface.

Chapter 1

Introduction

The goal of this chapter is to present the project that was proposed for the master's degree in Software Engineering at ISEP. The chapter starts with an initial context of the project as well as the problem to be solved and the main goals to be achieved.

1.1 Context

With the current demand for web-based applications, alternatives to REST continue to grow, one of them being GraphQL. The need for these alternatives comes from the problems within REST itself. Some of these problems are related to data-fetching (either over or under-fetching), the management of endpoints and the fact that REST endpoints are static, which can make them hard to maintain and version (Muench 2022).

GraphQL is an open-source project founded by Facebook in 2012 and "is a novel query language for implementing service-based software architectures" (Brito and Valente 2020). Although it started as an internal project, it has grown much more in popularity and is already adopted by multiple companies, such as Facebook, GitHub and Netflix.

1.2 Problem

GraphQL is a query language for APIs and runtime to provide data in response to those queries (*GraphQL | A query language for your API* 2022). It is possible to query only for the information you need, which is attractive, but some queries can lead to Denial of Service (DoS) (*GraphQL - OWASP Cheat Sheet Series* n.d.). These kinds of problems can be caused by querying large amounts of data or complex queries. If accepted, these queries will overload the system, and eventually cause DoS or extreme performance problems.

Most GraphQL APIs are vulnerable to either heavy performance problems or DoS through complex queries. Recognizing the necessity "to limit or prevent queries that are too expensive" (Mavroudeas et al. 2021) is fundamental, as well as trying to prevent, or at least improve, these concerns. Moreover, it is acknowledged that "practitioners lack principled means of estimating and measuring the cost of the GraphQL queries they receive" (Cha et al. 2020).

1.3 Objectives

The main objective of this thesis is to explore current approaches that try to minimize these concerns and identify their advantages and disadvantages by developing solution(s) while

acknowledging their impact on the system's performance. It is also important to understand when more robust approaches are required in comparison to the implementation of multiple simple approaches.

To achieve this, studies on how and why some GraphQL queries can be problematic and the various dimensions, how companies that use GraphQL deal with problematic queries and the different approaches that have been proposed to avoid DoS and their accuracy in limiting problematic queries, will be done as well as comparing the current approaches based on their capacity to correctly limit expensive queries. Analyze the main differences and their advantages and disadvantages. A study on the advantages of adopting multiple simpler approaches, instead of a single much complex approach, is also to be done.

Based on the analysis done, design one or more solutions based on current approaches that aim to make the use of GraphQL APIs more reliable. After these studies, the implementation of the designed solution(s) will be tried. These solution(s) will be evaluated based on the capacity to avoid malicious queries and the penalties on performance. The findings and the results recognizing what can be improved and addressed in future projects will be documented.

1.4 Research Methodology

The method used for the research was Design Science Research. This method is applied when the goal is to explore a new reality, like solving problems, instead of researching an already existing reality (Pello 2018).

This method generally has five stages (Figure 1.1), but it can differ depending on the project.

- **Awareness** of the problem and definition of the objectives
- **Design** of the proposed solution
- **Development** of the solution
- **Evaluation** of the solution, by comparing the results with the objectives
- **Conclusions** of the obtained results and how they can be relevant for future work.

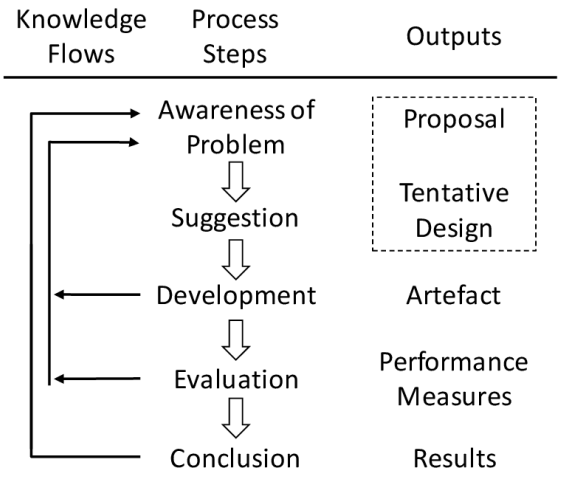

Figure 1.1: Design Science Research Methodology cycle (Beck, Weber, and Gregory 2013).

For this method is also necessary to define a research question. This question should be the focus of the research and should fully demonstrate what the project is trying to answer.

The defined question was: How can different query limiting strategies improve GraphQL performance and security concerns?

The approach followed to achieve the defined goals in 1.3 was split into the following stages:

- **Problem interpretation**: To fully understand the problem stated in 1.2 and identify the possible improvements.
- **Context**: Initial context on GraphQL.
- **State of the Art**: Analyse the current solutions that try to minimize the impact of the problem.
- **Design**: Design possible solution(s) that will try to solve or improve the problem.
- **Implementation**: Trying to implement the design solution(s).
- **Controlled experiments**: Testing the developed solution by forcing the problem in 1.2 and seeing the obtained results.
- **Result analysis**: Analyse the results obtained by the solution(s) implemented based on comparison with the current solutions. Identify the main advantages as well as possible limitations.

1.5 Structure

The structure defined for this document is as follows:

- **Introduction**: This chapter presents an initial view of the main problem and goals of the work done during this thesis. It also describes the approach used to achieve those goals. It ends with a quick explanation of the structure of the document.
- **State of the Art**: Presentation of GraphQL, security and performance concerns and the current solutions that try to solve the problem. It also shows a comparison between those solutions as well as a quick presentation of other technologies that were researched and used during the project.
- **Design**: This chapter presents the design choices decided for the implementation of the possible solutions.
- **Implementation**: Demonstration of the implementation of possible solutions.
- **Evaluation**: This chapter presents the tests of the proposed solution and result comparison with the existing technologies.
- **Conclusion**: The last chapter of this document describes the conclusions of the work done, the problems found during this work, the work to be done in the future and the final closing words.

Chapter 2

State Of the Art

2.1 GraphQL

GraphQL is a query language for Application Programming Interface (API)s founded by Facebook in 2012 and open-sourced in 2015. The idea for this project started back when Facebook was facing performance problems with their mobile applications. To solve this, there was a need for a new way to fetch data from their APIs (Figure 2.1). The solution encountered was to allow their applications to fetch only the data they needed.

Figure 2.1: *GraphQL | A query language for your API* 2022.

In 2019 the GraphQL Foundation was founded to manage the GraphQL project. Even though it started as an internal project for Facebook. Nowadays it's used by many big companies such as Twitter, GitHub and Shopify.

In the last few years, GraphQL has been growing in popularity, and one point of view is that shortly it could replace REST APIs. The replacement of REST entirely is not strictly necessary. Many projects are starting to combine the two alternatives to bring the best of both worlds. One usual approach is to use GraphQL as an abstraction for multiple REST APIs (Figure 2.2). This solution is especially useful when working with a microservices-based architecture, and GraphQL can provide an easy common gateway to access each service endpoint. This way we have one entry for all available endpoints and have the freedom for requesting the data needed.

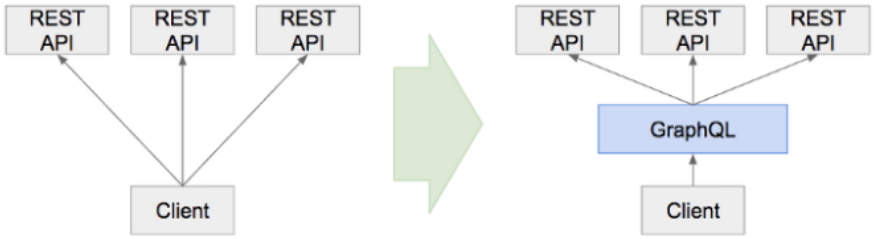

Figure 2.2: Microservices architecture before and after GraphQL gateway (Shtatnov and Ranganathan 2018).

The main benefits of using GraphQL for an API are: (Stuart 2018)

- **Performance** - Allowing to only get the data you need can improve performance by not over fetching data. Overfetching data happens when a GET request returns way more information than what is needed.
- **Flexibility** - The API does not define the size or shape of the data. So any possible user can define it according to their needs.
- **Evolution** - With REST APIs you can never delete current fields, this fields need to be deprecated. When deprecating a field you're not aware of clients consuming those fields. With GraphQL you can know which fields are fetched. With this information, you can make better decisions when deprecating fields.

Some disadvantages of adopting GraphQL can be (Shtatnov and Ranganathan 2018):

- **Selfish resolvers** - If you're using a GraphQL gateway architecture you need to be aware of possible duplicate requests. One possible solution would be to implement a caching solution (Shtatnov and Ranganathan 2018)
- **Pattern breaking** - Having the possibility to fetch partial information of an object breaks the Object Oriented Programming (OOP) paradigm.
- **Security** - The same flexibility that allows the user to fetch any data he wants can lead to potential Denial of Service (DoS). This security problem is the main focus of this project to explore ways of preventing these security issues.

Although GraphQL does not enforce it, there are defined guidelines that should be followed (Porcello and Banks 2018).

- **Hierarchical** - Like a graph, a GraphQL query is hierarchical. The requested fields are connected with each other.
- **Product centric** - The focus of GraphQL is to allow the client to specify the data he needs.
- **Strong typing** - In the GraphQL schema types need to defined according to the included type system in order to be validated.
- **Client-specified queries** - A GraphQL server provides the necessary information that a user is allowed to consume.
- **Introspective** - GraphQL is able to query its own type system.

2.1.1 GraphQL Characteristics

At its core, GraphQL is composed by:

- **Schema** where all the different types and interactions are specified. This is equivalent to the domain of our application and also the available operations (Source code extract 2.1 and 2.2).

  A type can be specified like:

```
type User {
    id: ID
    name: String
}
```

  Source code extract 2.1: Type definition.

  An operation can be specified like:

```
type Query {
  me: User
}
```

  Source code extract 2.2: Query definition.

- **Resolvers** where work is handled (Source code extract 2.3). "A resolver is a function that returns data for a particular field" (Porcello and Banks 2018)

```
const resolvers = {
    Query: {
        totalUsers: () => users.length
    }
}
```

  Source code extract 2.3: Resolver example.

In GraphQL there are three possible operations:

- **Queries** - An operation that allows to fetch data. Equivalent to a REST GET request (Source code extract 2.4).

```
{
  user(id: "1000") {
    name
  }
}

response:
{
  "data": {
    "user": {
      "name": "user1000"
    }
  }
}
```

Source code extract 2.4: Query example.

- **Mutation** - Allows to make changes or send data to the API. Equivalent to REST POST, PUT, PATCH or DELETE requests (Source code extract 2.5).

```
mutation CreateUser($name: String!) {
  createUser(name: $name) {
    id
    name
  }
}



variables:
{
  "name": "Hello"
}

response:
{
  "data": {
    "createUser": {
      "id": 1111,
      "name": "Hello"
    }
  }
}
```

Source code extract 2.5: Mutation example.

- **Subscriptions** - Event based operation to receive updates from the server (Source code extract 2.6).

```
subscription {
    statusChange {
        name
        status
    }
}
```

Source code extract 2.6: Subscription example.

2.2 Security

When developing a GraphQL API, some security concerns may be taken into account. Some of the most usual are:

- **Injections** - These can be database or operating system injections that execute unwanted code on the server-side. Or request injection by acting as a middle man between the client and the actual server.
- **Denial of Service (DoS)** - Requesting heavy quantities of data, ending in the server not being able to answer any request.
- **Abuse of authorization/default configurations** - Consists of taking advantage of either broken authorizations or default configurations that were not properly defined. A recent study found that most organizations either have only basic authentication or no authentication at all. With GraphQL in mind, some security breaches were found that could lead an attacker to extract sensitive data or even unauthorized transactions (Security Magazine 2021). This is especially worrying having in mind the context of fintech companies.

The main concern to be explored will be DoS since it's the most relevant for this project.

2.2.1 Denial of Service (DoS)

"Denial-of-Service attacks (DoS) are conducted against a running service to interrupt its availability to legitimate users" (Soliman and Azer 2019).

Denial of Service (DoS) attacks against GraphQL APIs consist of making queries that request large amounts of data or that have many nested queries. These attacks happen due to the freedom often given by these APIs to their users. In the hands of an attacker, this freedom can lead to intentionally sending these expensive queries to affect the system performance, and ending in an eventual Denial of Service. But it does not need to be a malicious user, a 'normal' user may be unaware of these problems and request data that can cause performance and DoS issues.

DoS still presents many challenges, some approaches are still embryonic (Suriadi, Clark, and Schmidt 2010). An attacker can use brute force and semantic attacks that will flood the system with unwanted heavy requests.

2.3 Possible Solutions

Although security concerns like DoS are one of the most important in GraphQL, some queries can still cause a heavy impact on a system without reaching the point of DoS. This is where the performance concerns also appear. It is safe to say that in many cases, performance issues happen more often. These concerns should be taken into account when analysing possible solutions.

As said before, one disadvantage that can come with adopting GraphQL is security. Because GraphQL allows the user to get the information he needs, a malicious actor could submit a very expensive query to overload our server, database and overall system (Stoiber 2018).

These queries can be expensive for two reasons:

- **Amount of elements requested** - When requesting a list, then the user can query for a very large number of results. This amount of data will overload the system.
- **Nested queries** - Nested queries can make the server compute a lot of internal requests to build the requested data.

An eventual attack on GraphQL APIs that allow this types of queries can lead to Denial of Service. Denial of Service means that eventually our server will crash and users won't be able to request the information they need.

To solve this possible problem GraphQL APIs need to make sure the received queries are safe, some solutions have been proposed.

They differ in the complexity needed for their implementation, the effectiveness and the possibility of affecting the API performance.

2.3.1 Size Limiting

This approach consists in limiting the size of the request. When the API receives a query it simply checks for its size and, depending on the defined limit, denies the request.

This approach is not very useful in a real-life scenario since it can create a lot of false positives. It can refuse simple queries that contain long names but allow expensive queries with short names that request a lot of information.

2.3.2 Query Whitelisting

This approach consists of having a list of allowed queries. By having this whitelist, any possible unwanted query wouldn't be possible since it's not allowed.

At first, this may look like a promising solution but there are some problems with it. The first is the maintenance of this list. Every time a new query was to be added, the list would need to be updated. This could take a lot of time and wouldn't be very practical. Luckily there are already solutions that extract the possible queries into a file (*GitHub - apollographql/persistgraphql: A build tool for GraphQL projects.* 2018). So this first problem would be solved.

A second problem with this approach is versioning. If a user is querying an older version of the API, he would need to be able to still send the queries that were available at that time. This means the developer can't remove any query or mutation from the list, only add. Once

again, the maintenance of this list would be complicated since it would require keeping a history of all the possibly queries.

The other problem is related to flexibility. GraphQL focus on the flexibility allowed to its users. By allowing only the queries in the list, the user is being restricted. In some cases, this may not be a problem if it's an API for internal use only. But if you're thinking of having a public API this approach is not ideal.

2.3.3 Depth Limiting

This solution, like the Size Limiting, consists in limiting the queries based on some criteria. In this case the number of nested queries.

This approach is pretty easy to understand and implement using external libraries. But it's still not enough to solve our problem. There's still the possibility to request a large amount of data. This limiting is useful but it should be implemented together with another solution.

2.3.4 Amount Limiting

Another limiting solution is to restrict the queries based on the amount requested. By doing this, the user wouldn't be able to request 10,000 elements of a list.

This solution can easily be achieved by using custom scalars instead of primitive types. So instead of having an integer as the number of elements to be requested, the value would have a custom number between a specific range.

As was stated above, this approach should be implemented together with a depth limiting solution and not a single approach.

2.3.5 Query Cost Analysis

With all these limiting solutions available, queries with an extensive number of elements or nested queries wouldn't be allowed. At this point, if the API is not complex, this could be enough. But for more advanced APIs, and to make sure it's fully safe from the possibility of DoS, there's still a lot of work to do.

One suggested solution to fully prevent this is the Query cost Analysis. This approach consists in calculating the cost of a query upon its arrival and deciding if it's safe or not to answer. This may look simple, but the work involved in implementing this strategy can be painful since it requires defining the cost for every possible element requested. Once again, if the API is extremely simple, this solution may not be worth implementing, but it's certainly the most secure out of the five presented.

For this solution there are already some approaches available like the validation complexity (*slicknode/graphql-query-complexity: GraphQL query complexity analysis and validation for graphql-js* n.d.), the cost analysis (*pa-bru/graphql-cost-analysis: A Graphql query cost analyzer.* N.d.) and the query complexity (*4Catalyzer/graphql-validation-complexity: Query complexity validation for GraphQL.js* n.d.).

Some techniques for this approach can be defining values for each type present in the GraphQL API. When executing a query, there are different costs for objects, scalars, enums, and also different costs for mutations compared to queries. This cost estimation should be the closest as possible to the real value, to prevent only malicious or expensive queries.

The main concern with this approach is performance. The query needs to be analysed in a way that doesn't affect the usual performance of the API. A very complex cost analysis that completely prevents expensive queries but affects the API performance, defeats the purpose of GraphQL.

2.3.5.1 GraphQL Validation Complexity

This approach consists of defining the value for each type present in the schema and based on the query sent, calculating what is the total cost of this query. This allows for a very easy implementation and an easy to adjust cost factor. By having this easy to use approach, each API developer can adjust the limits to his needs. By defining these limits outside the schema these values can be configured in a way that is easily accessible.

2.3.5.2 GraphQL Cost Analysis

This approach differs from the previous one since it allows us to define a cost directive directly in the schema. It also has the ability to have list multipliers, this means that the cost is influenced by the number of elements requested. This is the most complex approach and probably the most secure, but it comes with the cost of time to implement and also a harder way of adjusting the cost limits since they will have to be changed in the schema directly.

2.3.5.3 GraphQL Query Complexity

The query complexity approach follows the previous one, by allowing to set the cost directive but it does not support the list multiplier. As it follows the cost directive approach it also inherits its benefits, but not having the list multiplier is a major setback since it does not bring the also higher level of security with the more costly implementation.

2.3.6 Pagination

Pagination is a design pattern that consists in splitting data into pages of a specific size. Like this, the user is only allowed to fetch one page at a time and with the defined sizes. This approach is usually found on many web pages as well as search engines.

This pattern is usually useful in two scenarios:

- Loading/rendering the results takes too much time
- The results are "infinite"

In GraphQL these two scenarios are not usually applied, but in REST APIs they can happen more often. This solution brings several improvements to both performance and efficiency (Orlivskyi, Deomin, and Averianova 2021).

This approach limits the flexibility of the GraphQL client, who may not want to use pagination since it can introduce more complexity on both the server and client-side.

Some pagination techniques are:

- **Server-side pagination**. Server-side pagination is the more traditional by allowing the user to request a specific page of a specific size and return it to the client (Figure 2.3).

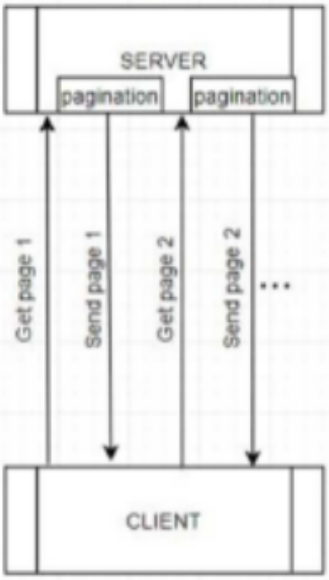

Figure 2.3: Server-side pagination (Orlivskyi, Deomin, and Averianova 2021).

- **Client-side pagination**. Where the client-side requests the full data and does the pagination itself. This can useful in small datasets. However, it's not recommended since it doesn't answer the problem with large quantities of data being requested to the server (Figure 2.4).

Figure 2.4: Client-side pagination (Orlivskyi, Deomin, and Averianova 2021).

- **Mixed pagination**. Mixed pagination is, as the name implies, a mixture between server

and client-side pagination. This is done by requesting paginated data from the server-side, but more than one page at a time. Then on the client-side, another pagination is applied on top. This helps reduce the requests from the client-side to the server-side without overloading the client with too much information (Figure 2.5).

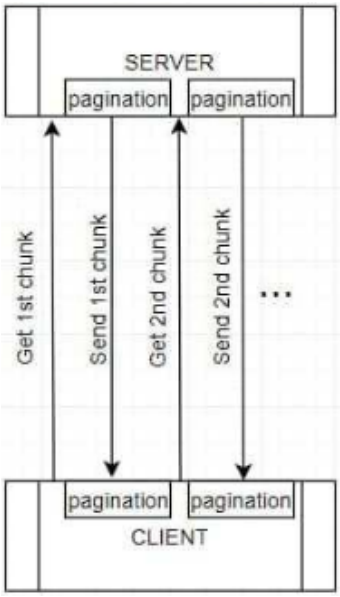

Figure 2.5: Mixed pagination (Orlivskyi, Deomin, and Averianova 2021).

2.3.7 Timeouts

This approach consists of refusing the request if it's not answered in the defined time range. Once again, this solution is also used in REST APIs and doesn't' fully answer GraphQL concerns.

Another problem would be how to define the correct value for this timeout. This value would need to be well defined to restrict only the real malicious requests. Even with this effort, by using timeouts, the API can limit legitimate users that send queries that may take more than the defined timeout.

2.3.8 Rate Limiting

Rate limiting consists of allowing a specific number of requests per user. This approach could be complex since it requires keeping a count on the requests per user or IP address. This solution can also block the users from requesting data even though no expensive queries were requested. In certain scenarios, this solution could be useful but in most cases is not ideal due to its unreliability in limiting only users with malicious intentions.

2.4 Practical Examples

Some companies that use GraphQL and document their findings are Netflix and Shopify. Both of them use some kind of Query Cost Analysis solution.

Netflix's approach is to have a static query cost calculation done for all incoming requests. This solution was implemented to avoid possible locks in the API gateway and their resources

(*Scaling Netflix's API via GraphQL Federation (2) | Netflix TechBlog* 2020). Besides that, for security concerns, both authentication and authorization were implemented.

Graph introspection, which is the ability to see the entire GraphQL schema is also restricted to only their internal development team.

Shopify's solution is to define values for each type of entity and operation present in their API. The calculation is done when receiving a request and calculating what will be the cost of the operation based on the defined values (*Rate Limiting GraphQL APIs by Calculating Query Complexity — Development* 2021).

Chapter 3

Value Analysis and Proposition

The goal of this chapter is to demonstrate the possible value of the proposed solution.

A value analysis consists of "evaluating the functionality of a product or a process in relation to its cost" (Mahalakshmi 2021).

In this case, the New Concept Development (NCD) model will be used to analyse possible opportunities and define their value and analyse it using the Analytic Hierarchy Process (AHP) method.

3.1 New Concept Development (NCD) Model

The front end of innovation (Figure 3.1) is one alternative to the Fuzzy front end and is the process of having an idea, developing that idea, testing it and commercialising it.

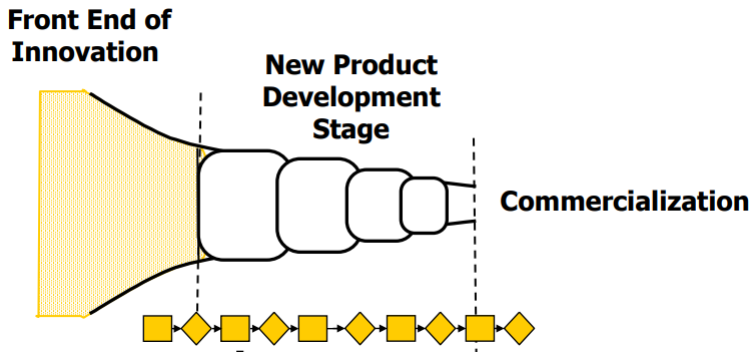

Figure 3.1: Front End of Innovation.

The Front End of Innovation (Figure 3.1) is the beginning stage of the innovation process. It's in this stage that the main opportunities are not only identified but analysed and validated.

By using the New Concept Development (NCD) Model (Figure 3.2), it allows us to have the key elements that define the Front end of Innovation of the project. This model has three main areas:

- The **engine** that drives the activities of the Front End of Innovation
- The **Five Front End Elements** of the NCD
- The influencing Factors that are uncontrollable

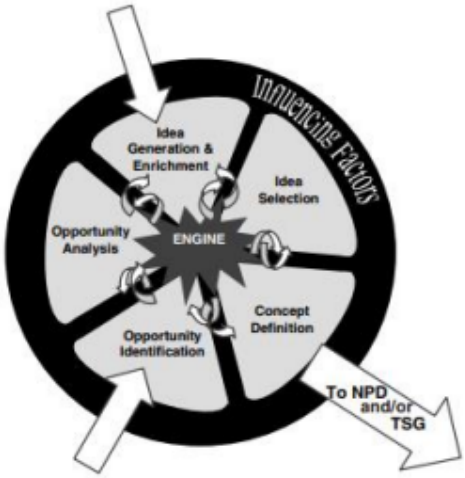

Figure 3.2: New Concept Development (NCD) Model.

Using the New Concept Development (NCD) Model, this are the five elements to explore:

3.1.1 Opportunity Identification

The growth of GraphQL opens the door for these vulnerabilities to happen more often. There's a need for these alternatives to be implemented to have safer and more secure APIs.

This way, GraphQL APIs can become more:

- Reliable
- Safer
- Effective
- Attractive to possible new teams who are thinking to adopt this technology

3.1.2 Opportunity Analysis

Although it may look like a promising idea, it's still necessary to further analyse it and answer the questions "against" this idea.

Some of these questions may be:

- How relevant is the concern about expensive GraphQL queries?
- Can my API maintain its performance even after these security measurements?
- Is it worth taking time to implement a difficult strategy instead of a simpler one?

To answer these questions, methods like AHP will be used to guarantee which is the best solution and how it brings value to teams who use GraphQL.

3.1.3 Idea Generation and Enrichment

Improvements in GraphQL expensive queries security can be done in multiple ways, some simpler like limiting the query based on the size, amount of elements and depth, some more complex like the use of query whitelisting or an "on-the-air" query cost analysis which can prevent this queries from being performed based on their cost.

3.1.4 Idea Selection

As said before the Analytic Hierarchy Process (AHP) method will be used to analyse and identify the best alternative having in mind the criteria defined. These attributes were chosen based on the identified problem and the main goal of the project. These attributes are:

- **Complexity**: How hard the solution is to implement, based on how long it would take approximately to implement
- **Performance**: What is the estimated performance of each solution and how it may impact its use, based on how long the query takes to be answered
- **Effectiveness**: The most important criteria is how each solution is estimated to be effective in solving the problem, based on problematic queries being accepted, leading to at least performance problems

After having the criteria defined the next step is to define alternatives to be analysed. In this project five alternatives were identified (Figure 3.3):

- Size Limiting
- Query WhiteListing
- Depth Limiting
- Amount Limiting
- Query Cost Analysis

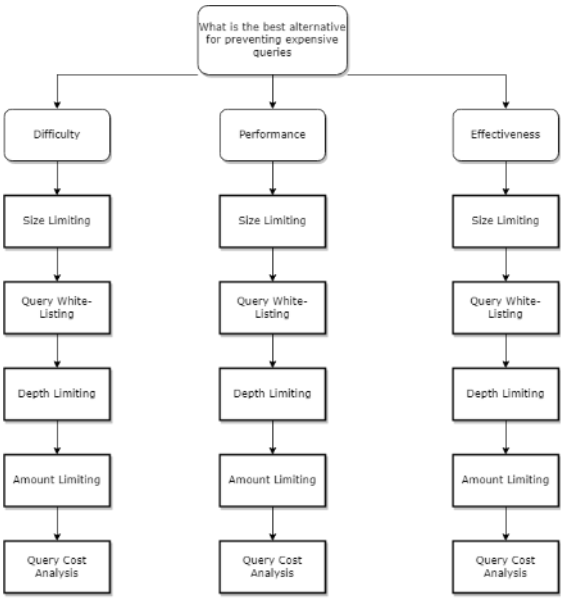

Figure 3.3: AHP criteria/alternative tree.

Now that the criteria and alternatives are defined, the next step is to define a comparison between the importance of each attribute. For this, the Saaty degree scale in the table 3.1 was used. The following values were assumed based on the knowledge of the theme prior to the actual implementation and in current studies.

Table 3.1: Saaty scale.

Scale	Definition
1	Equal importance
3	Slight importance
5	Strong importance
7	Very strong importance
9	Extreme importance
2,4,6,8	Intermediary values

Table 3.2: Criteria comparison.

	Complexity	Performance	Effectiveness
Complexity	1	1/3	1/5
Performance	3	1	1/3
Effectiveness	5	3	1

After having the comparison matrix (Table 3.2), the next step is to calculate the sum of each column (Table 3.3) and the priority vector for each attribute (Table 3.4)

Table 3.3: Criteria sum comparison.

	Complexity	Performance	Effectiveness
Complexity	1	1/3	1/5
Performance	3	1	1/3
Effectiveness	5	3	1
Sum	**9**	**13/3**	**23/15**

Table 3.4: Criteria priority comparison.

	Complexity	Performance	Effectiveness	Priority
Complexity	1/9	1/13	3/23	0.1062
Performance	1/3	3/13	5/23	0.2605
Effectiveness	5/9	9/13	15/23	**0.6333**

After calculating the priority vector, it is already evident that the most important attribute is effectiveness.

The next step is to validate the consistency of the matrix. This is done by calculating the consistency ration based on the consistency index and the random consistency index value (Table 3.5, Figures 3.4, 3.5 and 3.6).

Table 3.5: Random Consistency Index values.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

$$\lambda max = \Sigma(sum\ x\ priority) = (9\ x\ 0.1062) + (\frac{13}{3}\ x\ 0.2605) + (\frac{23}{15}\ x\ 0.6333) = 3.0557$$

Figure 3.4: Calculation of the max eigen value

$$CI = \frac{(\lambda max - n)}{(n-1)} = \frac{(3.0557 - 3)}{(3-1)} = 0.0279$$

Figure 3.5: Calculation of Consistency Index.

$$CR = \frac{CI}{RI} = \frac{0,0279}{0.58} = 0,0481 < 0.1$$

Figure 3.6: Calculation of Consistency Ratio.

Then, the next step is to define the comparison matrix of each alternative for each attribute (Tables 3.6, 3.7 and 3.8)

Table 3.6: Alternatives Complexity comparison.

Complexity	**Size Limiting**	**Query Whitelisting**	**Depth Limiting**	**Amount Limiting**	**Query Cost Analysis**	**Priority**
Size Limiting	1	3	2	2	5	**0.3661**
Query Whitelisting	1/3	1	1/3	1/3	3	0.1067
Depth Limiting	1/2	3	1	1	5	0.2384
Amount Limiting	1/2	3	1	1	5	0.2384
Query Cost Analysis	1/5	1/3	1/5	1/5	1	0.0504

Table 3.7: Alternatives performance comparison.

Performance	**Size Limiting**	**Query Whitelisting**	**Depth Limiting**	**Amount Limiting**	**Query Cost Analysis**	**Priority**
Size Limiting	1	1	1	1	2	**0.2222**
Query Whitelisting	1	1	1	1	2	**0.2222**
Depth Limiting	1	1	1	1	2	**0.2222**
Amount Limiting	1	1	1	1	2	**0.2222**
Query Cost Analysis	1/2	1/2	1/2	1/2	1	0.1111

Table 3.8: Alternatives effectiveness comparison.

Effectiveness	**Size Limiting**	**Query Whitelisting**	**Depth Limiting**	**Amount Limiting**	**Query Cost Analysis**	**Priority**
Size Limiting	1	1/3	1/5	1/5	1/9	0.0379
Query Whitelisting	3	1	1/5	1/5	1/7	0.0684
Depth Limiting	5	3	1	1	1/5	0.1651
Amount Limiting	5	3	1	1	1/5	0.1651
Query Cost Analysis	9	7	5	5	1	**0.5746**

The last step is to multiply the alternatives priority matrix by the attributes priority matrix to obtain the ideal solution (Figure 3.7).

$$\begin{Bmatrix} 0.3661 & 0.2222 & 0.0379 \\ 0.1067 & 0.2222 & 0.0684 \\ 0.2384 & 0.2222 & 0.1651 \\ 0.2384 & 0.2222 & 0.1651 \\ 0.0504 & 0.1111 & 0.5746 \end{Bmatrix} x \begin{Bmatrix} 0.1062 \\ 0.2605 \\ 0.6333 \end{Bmatrix} = \begin{Bmatrix} 0.1207 \\ 0.1125 \\ 0.1878 \\ 0.1878 \\ \mathbf{0.3982} \end{Bmatrix}$$

Figure 3.7: Best alternative calculation.

3.1.5 Concept Definition

After the analysis is done it is understood that the implementation of Query Cost Analysis solutions is the best alternative to improve the security and reliability of GraphQL APIs.

3.2 Perceived Value

To understand the real value of this project, it's necessary to identify the stakeholders and their benefits or disadvantage.

- **API developers**

  Benefits

  - Deliver a safer and more reliable product.

  Disadvantages

  - Need to take more time into implementing these strategies.
  - Not all developers may be aware of these solutions, therefore they may need some training.

- **Companies**

  Benefits

  - Deliver a safer and more reliable product.

  Disadvantages

  - Invest more time on making the product safe.

- **API users**

  Benefits

  - Having a more reliable source of data since its unlikely that expensive queries will break the API.

3.3 Value Proposition

To define the value proposition it's necessary to identify not only the product but also the gain creators and pain relievers, the gains, the pains and the customer jobs. This can be achieved by creating a value proposition canvas.

This value proposition presents the value that can be delivered by using a Query Cost analysis strategy in GraphQL APIs (Figure 3.8).

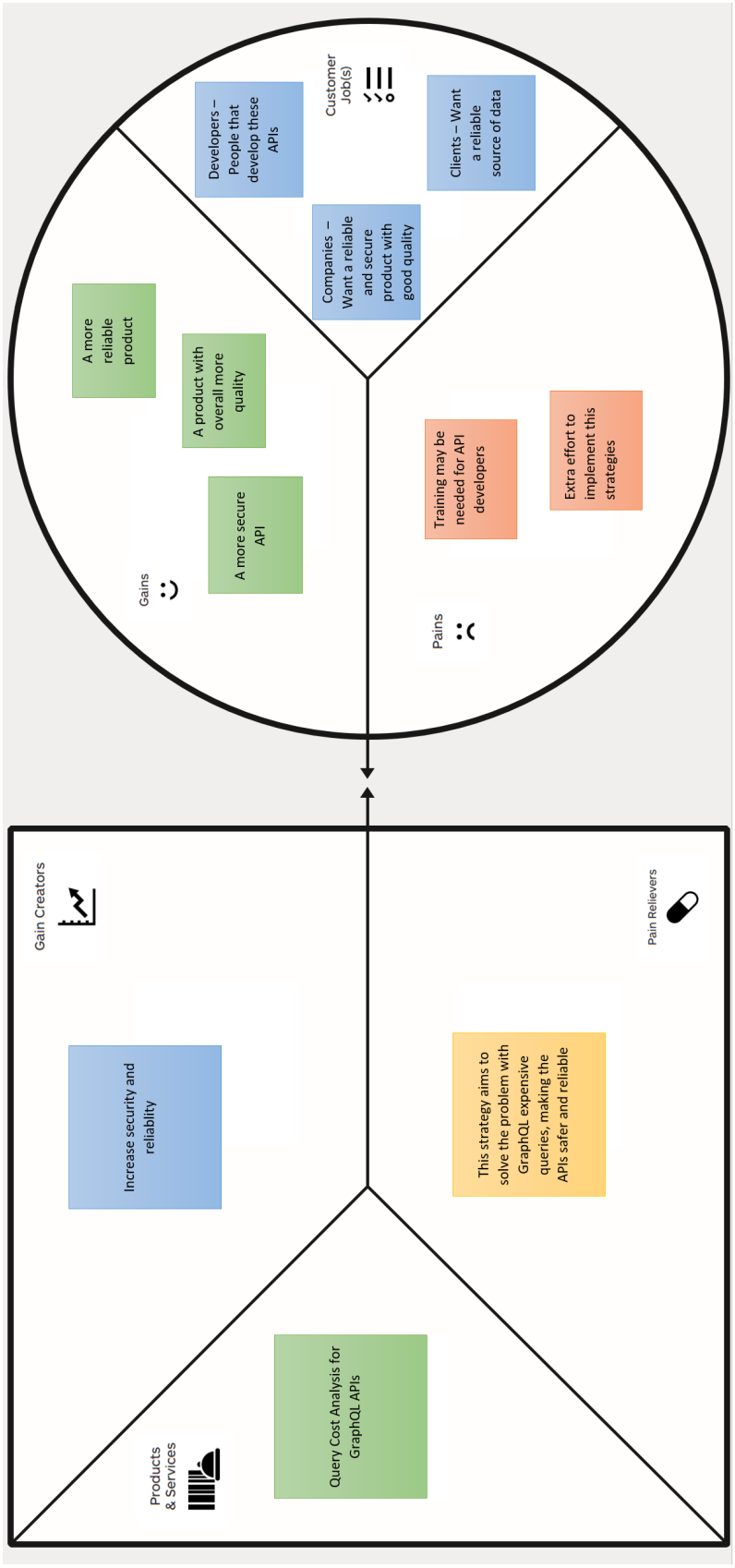

Figure 3.8: Value Proposition Canvas.

Chapter 4

Design

4.1 Project Selection Requisites

To explore the proposed solutions an already existing API will be used. This API needs to have specific requisites that guarantee the quality of the selected project and how it fits this thesis.

Some of the requisites of the select project are:

- The project needs to be public. Everyone reading this thesis should be able to access it.
- Have a minimum of lines of code so that small projects are filtered. A decent size project needs to be used, making sure the solution applied can improve or minimize the concerns.
- The GraphQL schema must be complex enough so that the performance and security issues can easily be forced. Alternatively, a less complex schema can also be selected and improved before the actual implementation.
- The project functionalities and performance must be maintained as much as possible.
- Use only open-source tools.

For this research, git repository hosting services, like GitHub and GitLab is to be used.

4.1.1 Chosen Project

The selected project was Emmy 2021. This project was selected due to the average project size and an easy-to-modify schema. Although in the beginning, the idea was to search for a project with a more complex schema and of a bigger size, this search turned out to be hard, both very small projects and very complex ones were found. In the end, the decision came to this project due to being in the middle in terms of complexity and familiarity with the language used (TypeScript).

The schema is to be improved to fully simulate the problem and test the solutions.

The name of the selected project is BOARD and contains both a GraphQL API and a UI. As will be explained in the next chapter the goal of this project is to provide a way of communication.

The main characteristics that make this project ideal to take are:

- Easy to improve domain

- Good initial structure
- Use of open-source tools
- Recent project creation (2021)
- Ease to get the project up and running. No need for many configurations that would consume important time for the study of the actual approaches to be implemented

4.2 Domain

The scope of the project is to create a platform for communicating between communities. The domain entities are all related to the scope.

The initial domain (Figure 4.1 and Appendix A.1) had five entities.

- **Announcement** - Represents an announcement to the community.
- **Suggestion** - Represents a suggestion that can be made by any user.
- **Post** - Represents a post shared with the community.
- **Trend** - Represents a something that is trending in the community.
- **View** - Represents a view of a trend, so it's easy to tell of viewed each trend.

The main problem of this domain is that the entities were mostly independent. This would mean that testing nested queries wouldn't be possible, so this was one of the focuses of the domain improvement.

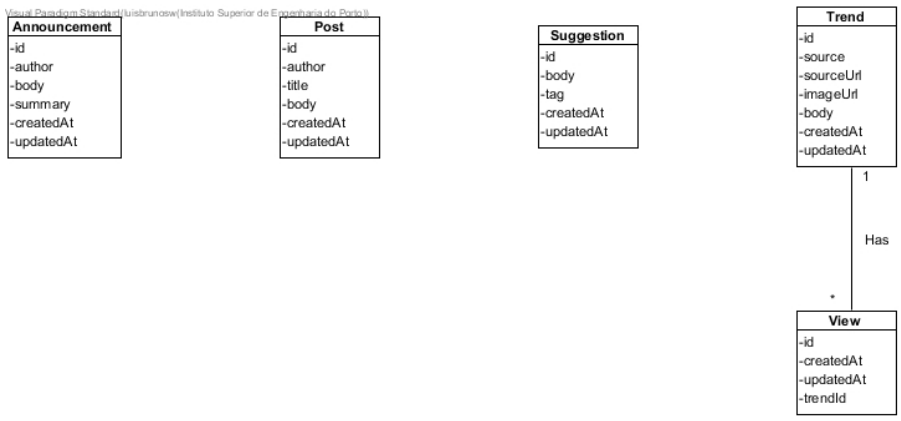

Figure 4.1: Initial domain

In the improved domain (Figure 4.2 and Appendix A.2) two new entities were introduced.

- **User** - Represents the user of the platform. This is how we can identify the community members.
- **Comment** - Represents a comment to a post. Comments can be made to posts in order for the community to interact.

The most important improvement wasn't this addition but creating connections between the already existing entities and these new ones. Now the entity user is connected to all the

other entities in order to identify the person who made any possible interaction. And the comment can be made to both a post or another comment.

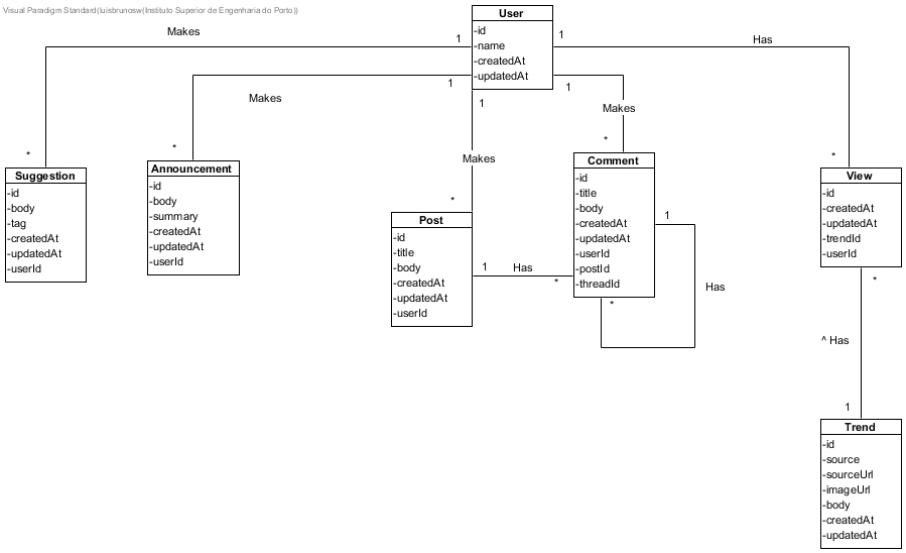

Figure 4.2: Improved domain

4.2.1 Architecture Design

This section aims to document the architecture designed to achieve the solution. For this, the C4 model (Brown 2022) was used. This model consists of having design diagrams for multiple abstraction levels. The further you go the more specific the diagram is.

4.2.1.1 Level 1 - System Contexts

In this level an overall system architecture must be designed (Figure 4.3 and Appendix B.1).

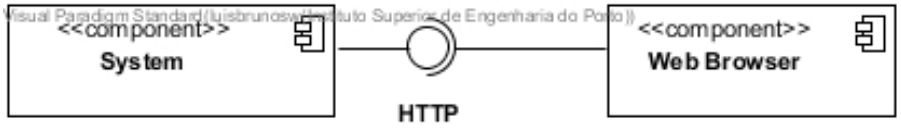

Figure 4.3: System Context Diagram

In this case we have two contexts:

- **Web Browser** - What the user(UI) will use to interact with the system.
- **System** - Our system itself, meaning this will by our whole application.

4.2.1.2 Level 2 - Containers

In the second level, the diagram aims to document the different containers of the system (Figure 4.4 and Appendix B.2).

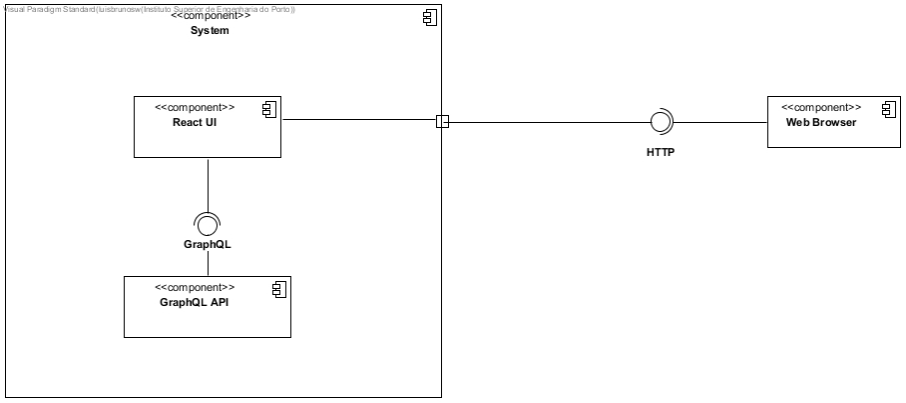

Figure 4.4: Container Diagram

In the System context we have two containers:

- **React UI** - How the user will interact with the API.
- **GraphQL API** - The GraphQL API where he can send queries and mutations.

4.2.1.3 Level 3 - Components

In the third level, there must be a diagram for each component of the system, in this case, there will be two components, the API (Figure 4.5 and Appendix B.3) and the UI (Figure 4.6 and Appendix B.4).

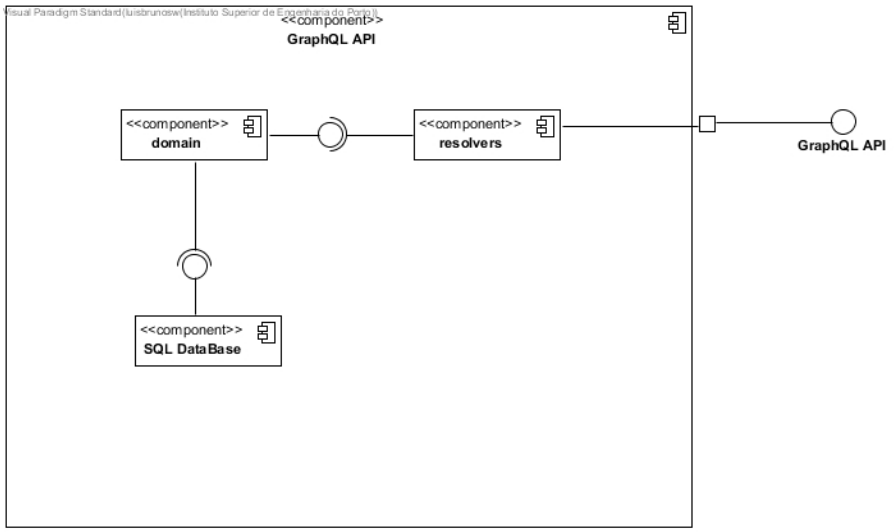

Figure 4.5: Component Diagram API

In the API container we have three components:

- **Resolvers** - Where the operations sent by the user will be resolved. This means it is where the system decides what to do with the request
- **Domain** - The domain of the application. Here we have the different entities referenced in the domain diagrams.

- **SQL Database** - This component refers to the database where the information will be stored.

This container has an entry being the GraphQL API.

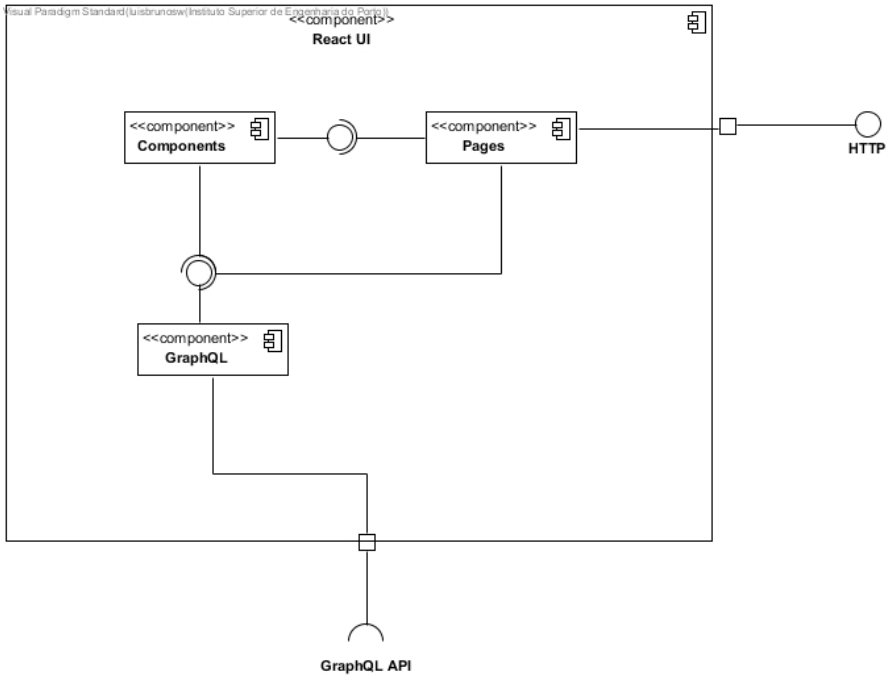

Figure 4.6: Component Diagram UI

In the UI container we also have three components:

- **Pages** - The different pages that can be seen in the UI.
- **Components** - The components used to create each page.
- **GraphQL** - This component is where the communication with the GraphQL API happens.

This container has an entry being the HTTP for the browser and consumes the GraphQL API.

Level 4 was not documented since it already refers to the code developed.

4.3 Requirement Analysis

The goal of the requirements is to document the needs of the stakeholders in a way it's easily comprehensible. To design the solution is fundamental to understand what are the requirements and the actors of the system.

4.3.1 Actors

In this case, the system actors are very easy to define. We have two different actors:

- API user, which can use the API directly

- UI user, which uses a User Interface to communicate to the API

4.3.2 Requirements

When documenting requirements, two types can be identified.

- Functional, that describe the main functionalities of the system
- Non-Functional, that are other requirements that should be taken into account but can't be qualified as a feature. Some these are:
  - Security
  - Performance

In this case the initial functional requirements found were (Table 4.1):

Table 4.1: Functional Requirements.

Id	Actor	Description
FR1	API	The user should be able to create an announcement (message)
FR2	API	The user should be able to get an announcement (message)
FR3	API	The user should be able to get announcements (messages)
FR4	API	The user should be able to create a view
FR5	API & UI	The user should be able to get views
FR6	API & UI	The user should be able to create a trend
FR7	API	The user should be able to get trends
FR8	API & UI	The user should be able to get new trends
FR9	API & UI	The user should be able to create a post
FR10	API	The user should be able to delete a post
FR11	API	The user should be able to get a post
FR12	API & UI	The user should be able to get posts
FR13	API & UI	The user should be able to create a suggestion (tip)
FR14	API & UI	The user should be able to get new suggestions (tips)
FR15	API & UI	The user should be able to get a suggestions (tips)

Additionally, after the schema was improved new requirements were also identified (Table 4.2):

Table 4.2: Additional Functional Requirements.

Id	Actor	Description
FR16	API	The user should be able to create an user
FR17	API	The user should be able to delete an user
FR18	API	The user should be able to get users
FR19	API	The user should be able to get an user
FR20	API	The user should be able to create a comment
FR21	API	The user should be able to delete a comment
FR22	API	The user should be able to get comments
FR23	API	The user should be able to get a comment

The Non-Functional requirements identified were:

- **Performance**, the solution must maintain stable and fast responses to the queries requested
- **Security**, Problematic queries shall not be accepted

The use case diagram can be found in Appendix C.1.

4.3.3 Requirements Design

This section documents the design of the functional requirements using Software Sequence Diagrams (SSDs).

Most of these diagrams are very similar, so only one example per operation will be presented GraphQL APIs written in TypeScript are very easy to understand since they require a simple structure. In the following diagrams, the classes present will be the resolver and the entity class.

4.3.3.1 FR1 - Create an Announcement

In this requirement the user wants to create an announcement, so he sends a mutation to do so. This mutation will be caught by the specific resolver, check if the author is valid and then create the announcement, returning it after (Figure 4.7 and Appendix C.2).

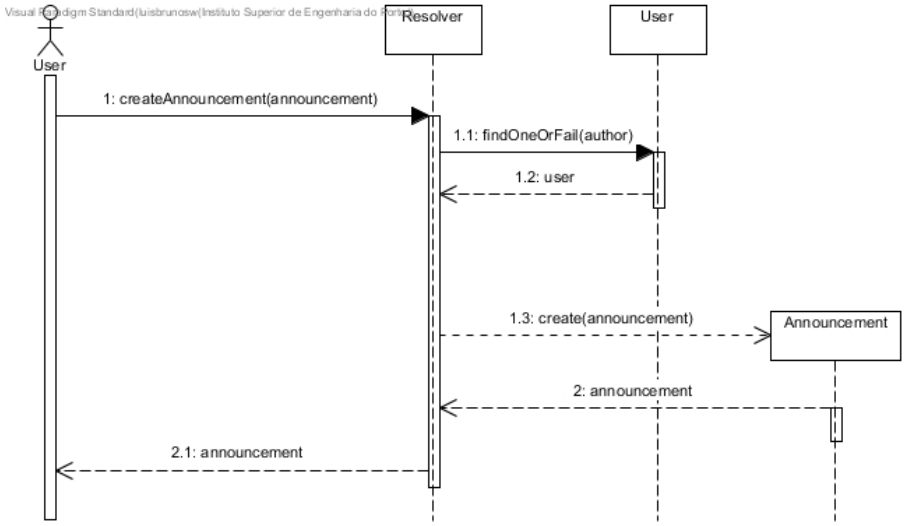

Figure 4.7: FR1 Sequence Diagram

4.3.3.2 FR10 - Delete a Post

In this requirement the user wants to delete a post, so he sends a mutation to do so. This mutation will be caught by the specific resolver and will try to delete the post with the post id that is sent in the mutation, if the deletion succeeds a boolean true is returned (Figure 4.8 and Appendix C.3).

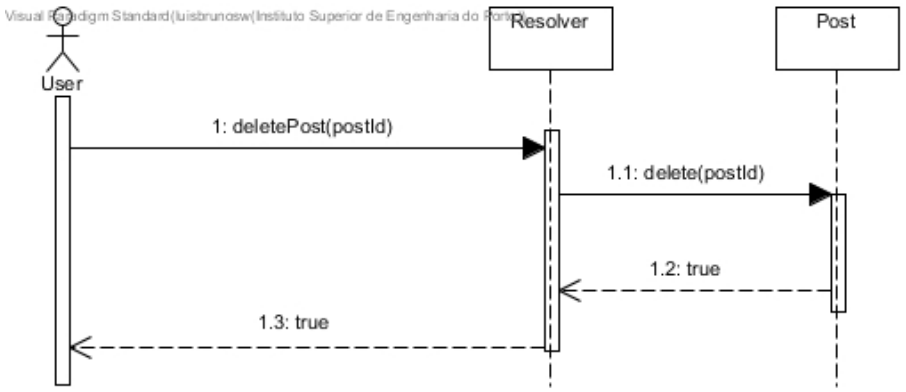

Figure 4.8: FR10 Sequence Diagram

4.3.3.3 FR19 - Get an User

In this requirement the user wants to get a user, so he sends a query to do so. This query will be caught by the specific resolver, and try to find the user with the id sent in the query, if found it will return it, otherwise an error will occur (Figure 4.9 and Appendix C.4).

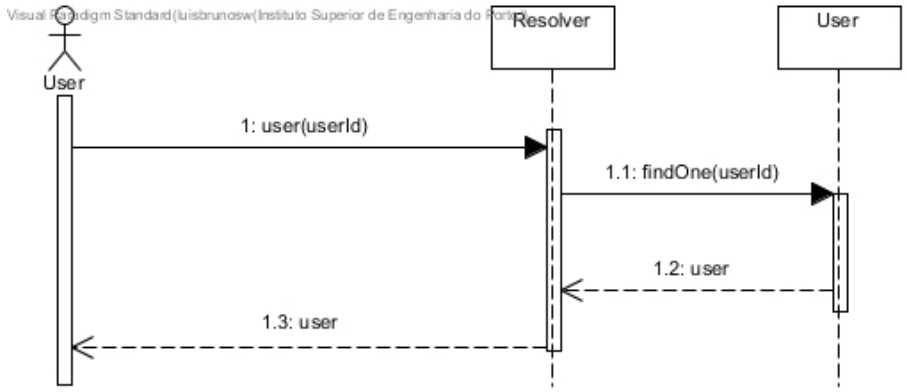

Figure 4.9: FR19 Sequence Diagram

4.3.3.4 FR22 - Get Comments

In this requirement the user wants to get comments, so he sends a query to do so. This query will be caught by the specific resolver and find all comments possible, returning what was found (Figure 4.10 and Appendix C.5).

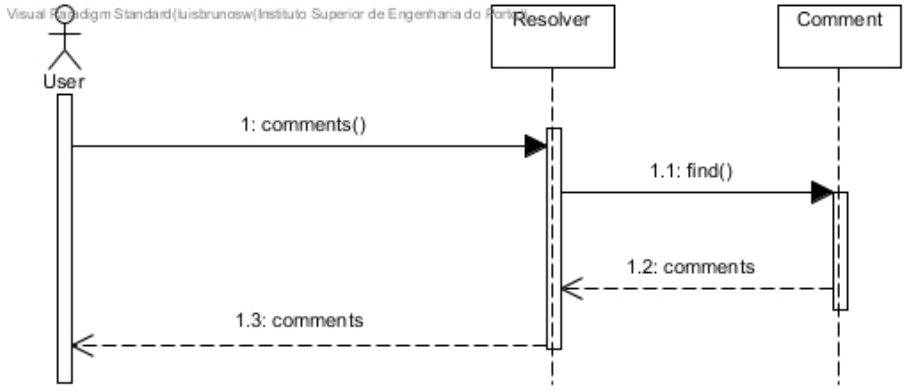

Figure 4.10: FR22 Sequence Diagram

Chapter 5

Implementation

In this chapter, details of the implementation are to be explained based on the design documented before.

The implementation contains the following steps and can be found in the public repository (Monteiro 2022).

- **Base Implementation (Prototype)** - Improvement of the exiting API to have a more complex domain and refactor some key areas to make the API more understandable.
- **Alternative 1** - Implementing multiple limiting techniques to the base implementation.
- **Alternative 2** - Implementing Query Cost Analysis to the base implementation.
- **Alternative 3** - Trying to create a joint solution.

5.1 Base Implementation

In the base implementation, the focus was to make minor changes so that the API was more cohesive and so we could improve the schema.

5.1.1 Schema

In this project, there is no need to create the schema by hand. The schema is automatically generated by looking into the domain and the specified resolvers. This uses the build schema method from the type-graphql library (Lytek 2020). The initial schema was generated and the queries (Source code extract 5.1) and mutations (Source code extract 5.2) correspond exactly to what was pretended.

```
type Query {
  posts: [Post!]!
  post(postId: String!): Post!
  announcements: [Announcement!]!
  announcement(annId: String!): Announcement!
  suggestions: [Suggestion!]!
  newSuggestions: [Suggestion!]!
  hello: String!
  trends: [Trend!]!
  newTrends: [Trend!]!
  views(trendId: String!): String!
}
```

Source code extract 5.1: Initial schema queries.

In the schema queries, we have

- **The name of the query** - In this case the name of the entity that is to be queried, for example, "posts".
- **The return** - The type of return of the query, in most cases, will be the type of the entity, so for the query "posts" the return will be a list of posts.
- **Input** - The input is present in some queries, usually to indicate what is the specific id we want to query.

In the query type the "!" character means that type is mandatory and the "[]" means the type is a list.

```
type Mutation {
  createPost(input: CreatePostInput!): CreatePostResponse!
  deletePost(postId: String!): Boolean!
  createAnnouncement(input: CreateAnnInput!):
   CreateAnnResponse!
  createSuggugestion(input: CreateSuggInput!):
   CreateSuggResponse!
  createTrend(
    body: String!
    imageUrl: String
    sourceUrl: String
    source: String!
  ): Trend!
  createView(trendId: String!): Boolean!
}
```

Source code extract 5.2: Initial schema mutations.

Similar to the queries, in the mutations we have

- **The name of the mutation** - Usually it will be the name of the action followed by the name of the entity, like "create" + "Post".

- **The return** - The type of return of the mutation, in the "create" action the return is mostly of a custom type that returns that created entity and the status of the response. If the action is a "delete" it just returns a boolean to indicate the success or not of the operation.
- **Input** - Like the return, the input change between creation and deletion. In the creation, the input will be either a custom type with the information needed to create the entity or the full attributes that need to be sent. In the deletion, it works like the query by id where the id field is the only information that needs to be sent.

Like in the queries, the "!" character means that type is mandatory and the "[]" means the type is a list.

With the new entities of user and comments added to the initial domain both the queries (Source code extract 5.3) and mutations (Source code extract 5.4) suffered changes and the new entities were also created (Source code extract 5.5).

```
type Query {
  posts: [Post!]!
  post(postId: String!): Post!
  comments: [Comment!]!
  comment(commentId: String!): Comment!
  users: [User!]!
  user(userId: String!): User!
  announcements: [Announcement!]!
  announcement(annId: String!): Announcement!
  suggestions: [Suggestion!]!
  newSuggestions: [Suggestion!]!
  hello: String!
  trends(input: TrendFilter): [Trend!]!
  newTrends: [Trend!]!
  views(trendId: String!): String!
}
```

Source code extract 5.3: -Improved schema queries.

In the query type, the only visible improvement is the new queries generated for the introduced entities.

```
type Mutation {
  createPost(input: CreatePostInput!): CreatePostResponse!
  deletePost(postId: String!): Boolean!
  createComment(input: CreateCommentInput!):
   CreateCommentResponse!
  deleteComment(commentId: String!): Boolean!
  createUser(input: CreateUserInput!): CreateUserResponse!
  deleteUser(userId: String!): Boolean!
  createAnnouncement(
    input: CreateAnnouncementInput!
  ): CreateAnnouncementResponse!
  createSuggugestion(input: CreateSuggestionInput!):
   CreateSuggestionResponse!
  createTrend(input: CreateTrendInput!): CreateTrendResponse!
  createView(input: CreateViewInput!): CreateViewResponse!
}
```

Source code extract 5.4: Improved schema mutations.

Unlike the queries, in the mutations, there was an improvement to homogenize both the inputs and returns.

```
type User {
  id: ID!
  name: String
  comments: [Comment]!
  posts: [Post]!
  views: [View]!
  announcements: [Announcement]!
  suggestions: [Suggestion]!
  createdAt: DateTime!
  updatedAt: DateTime!
}
```

Source code extract 5.5: User entity schema.

The custom schema types for the entities work like the queries and mutations. So they have

- **Name of the attribute** - This are the fields that can be queried.
- **The type of attribute** - This can be scalars like "String" and "DateTime" or custom types that reference other entities like the "Post" or "Comment"

Just like the queries and mutations, the "!" character means that type is mandatory and the "[]" means the type is a list.

5.1.2 Code

As documented in the design chapter, the API consists of having entities and resolvers. Here the focus will be the "Comment" entity (Source code extract 5.6).

```
@ObjectType()
@Entity("comments")
export class Comment extends BaseEntity {
  @Field(() => ID)
  @PrimaryGeneratedColumn("uuid")
  id: string;

  @ManyToOne(() => User, (user) => user.id)
  @JoinColumn({ name: "userId" })
  author: string;

  @Field()
  @Column({ type: "varchar" })
  title: string;

  @Field()
  @Column({ type: "text" })
  body: string;

  @ManyToOne(() => Post, (post) => post.comments)
  @JoinColumn({ name: "postId" })
  post: string;

  @Field(() => [Comment], {nullable: "items"})
  @OneToMany(() => Comment, (comment) => comment.id)
  answers: Comment[];

  @ManyToOne(() => Comment, (comment) => comment.answers)
  @JoinColumn({ name: "threadId" })
  thread: string;

  @Field()
  @CreateDateColumn()
  createdAt: Date;

  @Field()
  @UpdateDateColumn()
  updatedAt: Date;
}
```

Source code extract 5.6: Comment entity.

In the entity files, there are two main aspects,

- **Attributes** - The fields of the entity and their types.
- **Annotations** - In this implementation the annotations are very important for schema generation. As said before the schema is generated automatically based on the entities and resolvers, to do so we rely on annotations to identify multiple characteristics of

the schema. The annotations are split into two types, the GraphQL annotations and the database annotations.

GraphQL annotations:

- **ObjectType** - Identifies this class as a custom schema type.
- **Field** - Identifies the attribute as a schema field. If an attribute does not have this annotation it cannot be selected in the queries like the "post" field above. In this annotation, we can also specify what is the type of field in the schema. In the case above we have the "ID" meaning the field will be identified as the id of the entity and also the type of field when the field is supposed to be a list of a different database like the "answers" field where the return will be a list of type "Comment". In the field annotation, we can also have many extra properties like the "nullable" where in this case we define that the list of answers can be empty.

Database annotations:

- **Entity** - Used to identify the class as a database entity and what is the table to look for.
- **PrimaryGeneratedColumn** - This means this attribute will be automatically generated as a "UUID" value in the database.
- **Column** - In the column annotations, we can have multiple types, including the column itself which means the field is a column in the database. The JoinColumn means this field is the key to join a column from a different database. With JoinColumn we also need an extra annotation which is the relation. The relation can be ManyToOne which means our entity connects to a single entity of the other database. OneToMany means our entity connects to multiple entities of the other database. Besides this, we also have the CreateDateColumn and UpdateDateColumn which are a type of columns that will be filled automatically with the dates of creation and update of the entity.

Next, in the resolvers we all the different actions that can be performed in the API, this meaning the queries and mutations. Like the entity, the resolver class will also rely on annotations, like the "Resolver" annotation at class level, for the schema generation (Source code extracts 5.7, 5.8 and 5.9).

```
@Query(() => [Comment])
async comments(): Promise<Comment[]> {
  return await Comment.find({ order: { createdAt: "DESC" }
 });
}
```

Source code extract 5.7: Comments query resolver.

To create a query we must annotate the method with the query annotation and specify its return. The actual implementation is straightforward, in this case, the comments query will just call the "find" method in the "Comment" entity which will bring all comments in this case ordered by the date o creation.

```
@FieldResolver(() => User)
  async author(@Root() comment: Comment): Promise<User> {
    return await User.findOneOrFail(comment.author)
  }
```

Source code extract 5.8: Comment field resolver.

The FieldResolver annotation is used to, as the name says, resolve a specific field. In this case, if the user queries for the author of the comment, we need to fetch this information from a separate database. This resolver is used exactly to tell how can we fetch this field. Besides the FieldResolver annotation, we can specify the Root meaning what is the comment that we want to know the author. Like the query resolver, the field resolver, in this case, will fetch the user with the author sent and if not found throw an error.

```
@Mutation(() => CreateCommentResponse)
  async createComment(
    @Arg("input") { author, body, title, post, thread }:
   CreateCommentInput
  ): Promise<CreateCommentResponse> {
    const user = await User.findOneOrFail(author);
    const postOrThread = await this.getPostOrThread(post,
   thread);

    const comment = await Comment.create({
      ...postOrThread,
      author: user.id,
      title,
      body,
    }).save();

    return {
      ok: true,
      comment,
    };
  }
```

Source code extract 5.9: Create comment resolver.

The mutations are similar to the other two. We must annotate the method with the Mutation annotation and specify its return. Because this is a creation we also need the input. In this case, both the input and output were separated from the resolver to have a more cohesive resolver. Because we have fields from other databases, in the creation we will validate if the entity in those databases exists. If it does then we proceed, if not it will throw an error. If all the entities are found we try to create the entity with the required information and return the success of the operation.

The input and output of the mutation were designed in a way that is similar to all entities (Source code extracts 5.10 and 5.11).

```
@InputType()
export class CreateCommentInput {
  @Field()
  author: string;

  @Field({nullable: true})
  post: string;

  @Field()
  title: string;

  @Field()
  body: string;

  @Field({nullable: true})
  thread: string;
}
```

Source code extract 5.10: Create comment input.

The input has the annotation InputType to indicate to the schema that this is an input and, like the entity, the Field annotation to indicate what are the fields that can or must, be in the input.

```
@ObjectType()
export class CreateCommentResponse {
  @Field()
  ok: Boolean;

  @Field(() => Comment)
  comment: Comment;
}
```

Source code extract 5.11: Create comment output.

The output is annotated with ObjectType because it needs to be identified as a type in the schema and once again the Field to indicate what are the fields returned.

Besides the domain and resolvers, the other relevant characteristics of the API are the use of a PostgreSQL database and ApolloExpress server.

5.2 Alternative 1 - Limiting Approach

The first alternative consists of implementing more than one limiting approach. To do so, both depth and amount limiting was used. In the schema, the only visible change is the list filter (Source code extract 5.12).

```
type Query {
  posts(filter: ListFilter!): [Post!]!
  post(postId: String!): Post!
  comments(filter: ListFilter!): [Comment!]!
  comment(commentId: String!): Comment!
  users(filter: ListFilter!): [User!]!
  user(userId: String!): User!
  announcements(filter: ListFilter!): [Announcement!]!
  announcement(annId: String!): Announcement!
  suggestions(filter: ListFilter!): [Suggestion!]!
  newSuggestions: [Suggestion!]!
  hello: String!
  trends(filter: ListFilter!): [Trend!]!
  newTrends: [Trend!]!
  views(trendId: String!): String!
}
```

Source code extract 5.12: Query schema with list filter.

5.2.1 Amount limiting

The amount limiting approach consists of having a limit on how many items the user can request inside a list. For this, a new "InputType" was created (Source code extract 5.13).

```
@InputType()
export default class ListFilter {
  @Field(() => PaginationAmount)
  first: typeof PaginationAmount
}
```

Source code extract 5.13: List Filter input.

Initially, the "first" attribute was intended to be of type number but like this, the user would still be able to fetch large amounts of data. With a custom type like "Pagination-Amount", we can further limit the amount by allowing only the values we pretend. In the "PaginationAmount" (Source code extract 5.14) we can define whatever rules we pretend.

```
const PaginationAmount = createIntScalar({
    name: "PaginationAmount",
    minimum: 1,
    maximum: 100
  });
```

Source code extract 5.14: Pagination Amount scalar.

In this case, the pagination amount will have a minimum and maximum value, meaning the request list items will have to be between these values, this means that the minimum value asked is one element and the maximum, one-hundred elements. To create this custom scalar the library graphql-scalar (Cho 2019) was used.

In the domain there's no change, the only change needed is to add the custom input type to all the queries we want (Source code extract 5.15).

```
@Query(() => [Comment])
  async comments(@Arg("filter") {first}: ListFilter): Promise
   <Comment[]> {
    return await Comment.find({ take: parseInt(first.toString
   ()), order: { createdAt: "DESC" } });
  }
```

Source code extract 5.15: List Filter query resolver.

To do this, we only need a new attribute with the annotation "Arg" meaning this is an argument from the query. The value sent is then passed to the database find method.

5.2.2 Depth Limiting

For the depth limiting, typescript already offers that possibility in the "types" library. To do so, we just need to import it and add it to the Apollo validation rules (Source code extract 5.15).

In the domain there's no change, the only change needed is to add the custom input type to all the queries we want (Source code extract 5.16).

```
validationRules: [ depthLimit(5) ]
```

Source code extract 5.16: Depth Limit.

What the depth limiting function does is take the value that is passed to it, in this case, five, and when the query has more than five nested levels it will throw an exception.

5.3 Alternative 2 - Query Cost Analysis Approach

In the second alternative, the approach is to use query cost analysis to validate if the query is safe to handle. The choice went to graphql-validation-complexity since it's very simple to implement and in most cases, it will be more than enough.

To do this we just need to import the method createComplexityLimitRule from the library and then specify what should the factor for each type (Source code extract 5.17). We can also define custom values if we think a specific type requires an extra cost. In this case, the solution was to go for a cost of 1 for scalar types, 5 for objects and 10 for lists. The main thing missing with this solution is the ability to multiply the cost based on the number of elements in the list.

```
validationRules: [
      createComplexityLimitRule(20000, {
        scalarCost: 1,
        objectCost: 5,
        listFactor: 10,
        formatErrorMessage: (cost) =>{
          return `Query exceeds complexity limit. Calculated
   Cost: ${cost}`
        },
        onCost: (cost) =>{
          console.log(`Calculated Cost: ${cost}`)
        }
      })
    ],
```

Source code extract 5.17: Query cost analysis.

Like the depth limit, the query cost analysis goes into the validation rules, where we can implement many more other rules if necessary.

Under the hood what the method does is give each type the factor specified and when a query is received, it detects all the types present and calculates what would be the cost. In this case, if the cost exceeds the value of 10,000 the query is rejected with the message defined above.

5.4 Alternative 3 - Joint Approach

The third and last alternative consists of combining both the alternatives 1 and 2. This means we can easily reject queries that exceed the defined value, but will also reject queries that may escape that threshold if there are still large amounts of data requested. Depending on the project both the complexity and depth limit should be adjusted to further limit more queries or let more queries in.

The final schema and folder structure can be found in Appendices D and E

Chapter 6

Evaluation

The chapter of evaluation and experimentation has the goal to validate the solution presented and validate that it solves the presented problems.

For these validations, it is necessary to define specific measurements that will be analysed and evaluated.

6.1 Measurements

The definition of these measurements is essential to guarantee the quality of the solution.

Based on the proposed objectives, the attributes to be analysed are:

- **Quality of the implementation**. The presented solutions must be filtered to only the ones that don't have any implementation issues. The use of solutions with implementation problems may influence the results.
- **Performance**. The main measurement based on the problem is exactly the performance. It is essential to find which solutions present a better performance and how it compares to the original project. In this case, extreme measurements like the eventual DoS must be also measured.

  To analyse the performance, different tests will be done. The first is related to the time the solutions take to answer. And secondly the number of times the solutions reject the queries.
- **Problematic Query rejection**. How accurate the solutions are in rejecting problematic queries.

These measures will help evaluate each solution and find what is the best solution in different cases.

6.2 Evaluation Methodology

For the evaluation, five steps were followed.

- Exploration
- Hypotheses Construction
- Experiment
- Analysis

- Conclusions

It's important to know that these steps work in a cycle, once the conclusions are obtained, the method can reiterate to explore the problem again, or construct new hypotheses.

This process started with the exploration of what could be the problem and gathering data. Then there was an analysis during the problem definition, proceeded by a study in the state of the art. After the study, the hypotheses were created and followed by their evaluation. Finishing in with the conclusions on the evaluation done.

6.2.1 Testing scenarios

For this problem, the testing scenarios constructed were:

- **Scenario 0** - GraphQL API does not show any sign of performance issues when requesting problematic queries. Base Implementation of the prototype.
- **Scenario 1** - Comparison between Query limiting solutions (Alternative 1) and Query Cost analysis (Alternative 2).
- **Scenario 2** - Comparison between Query limiting solutions (Alternative 1) and the combination of both alternatives (Alternative 3).
- **Scenario 3** - Comparison between Query Cost analysis (Alternative 2) and the combination of both alternatives (Alternative 3).

6.2.2 Evaluation

To evaluate the alternatives and discover which are the correct testing scenarios, we need to have a query that is complex enough to be considered problematic. In this case, the query used was to get the users and all the information about them, as well as multiple nested fields. Even though this query could have been built manually, it was automatically generated by Postman. Postman is an API testing application that allows us to send requests to our GraphQL API. By importing our schema into postman, it already generates many possible queries that we can send. This means that a malicious user does not even need to build a query himself and generate these automatic queries to try to cause problems to APIs. The query used is too large to include in this document but it can be easily found in the repository (Monteiro 2022) in each "server" folder with the name "testQuery". Having in mind that the test queries were run with small amounts of data in the database, this means that any problem possible found is mostly due to the API and not to the amount of data in the database.

6.2.2.1 Scenario 0 - Base Implementation

To start the evaluation, we need to first validate that the problem actually exists by rejecting the alternative 0.

When we execute the test query into the base implementation API we understand that the problem is indeed present. The query is not answered and in the end, we get a heap error (Figure 6.1) this means the server does not have any more memory to allocate to resolve this query. This error means that in a real-world scenario this query would take down the API, so we are facing a possible DoS attack.

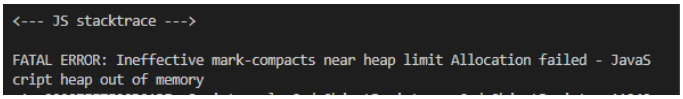

Figure 6.1: Heap exception

Besides the error, we can also see that the time it took to return the error was very long (Figure 6.2) (580,192ms which means approximately 9 minutes). So before the actual memory problem, the API would already have performance problems.

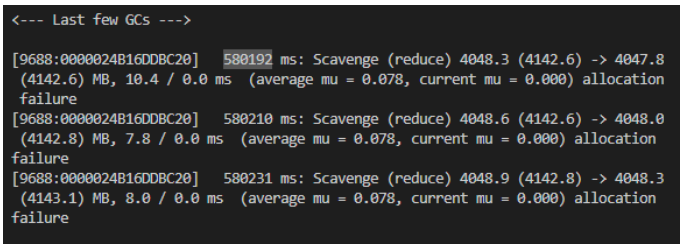

Figure 6.2: Time to occur the error

If we try with a smaller query (Source code extract 6.1) we can see that is answered and in a time that is still acceptable (Figure 6.3).

```
query users {
    users {
        id
        name
        comments {
            title
            body
        }
        posts {
            title
            body
            comments {
                title
                body
                author {
                    name
                }
            }
        }
        announcements {
            body
            summary
        }
        suggestions {
            body
            tag
        }
    }
}
```

Source code extract 6.1: Alternative 2 accepted test query.

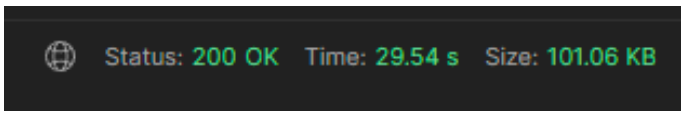

Figure 6.3: Accepted query time

6.2.2.2 Alternative 1

Starting now with the evaluation of the alternatives, we start with alternative 1.

In the alternative 1 where the limiting approach was implemented, we cannot request as much data as in the base implementation. So the maximum value of 100 will be used. As we can see in Figure 6.4 the limiting approach does not allow the API to have a memory error. But the time to answer the query is too long to be acceptable.

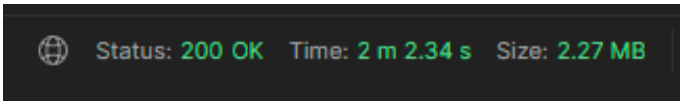

Figure 6.4: Time to answer test query in alternative 1

By looking into the defined measurements we can see that

- **Quality of the implementation**. The implementation of the size limiting only adds a limit to the number of elements being requested so problems with implementation are not relevant. The depth limiting is an external library but in this case, it does not have any impact since the test query is inside the defined limit.
- **Performance**. As we can see, the performance of this alternative is deeply affected by problematic queries. The test query does not have the max limit depth and still causes problems in response times. The implementation does not affect the performance since it does not add any additional effort from the server.
- **Problematic Query rejection**. The query rejection only happens when the maximum number of elements or nested elements are requested. Although the test query complies with the limits we can still see performance problems. These performance problems escalated to a bigger scale will lead to heavier performance issues.

6.2.2.3 Alternative 2

To evaluate alternative 2, the same logic will be followed. So we execute the query and the result is immediately an error (Figure 6.5). As we can see the calculated cost is way above the defined limit of 20,000.

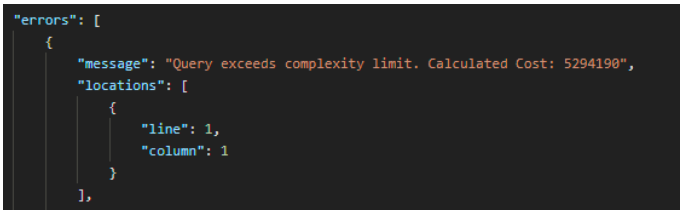

Figure 6.5: Calculated cost

When we try with a smaller query (Source code extract 6.1) we can see that it is accepted (Figure 6.6). The value under the 30s would still be considered acceptable but depending on the API and scenario, the limit value should be adjusted for a more restricted acceptance.

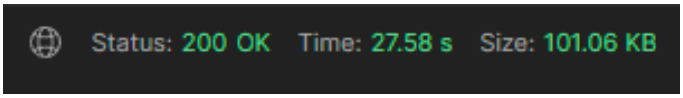

Figure 6.6: Query Cost Analysis accepted query time

Looking to the measurements again:

- **Quality of the implementation**. Once again the implementation does not have any relevant quality issue since it implements an external library that is vastly used.

- **Performance**. The performance of the queries is not affected. As we can see the time to answer the smaller query is almost the same, sometimes being even faster.
- **Problematic Query rejection**. The capacity to reject queries is significantly better than alternative 1 since it blocks the query as it knows the calculated cost. The main problem as said in the implementation chapter is that we can still request large amounts of data without it affecting the calculated cost. The cost is based on the types requested and not on the number.

By analysing the measurements we can already tell that alternative 2 can be considered better than 1, this evaluation impacts testing scenario 1.

6.2.2.4 Alternative 3

The last evaluation is on alternative 3, so the combination of alternatives 2 and 3.

When we request the more expensive query, as it is based on alternative 2 it is immediately rejected, so no changes in this aspect. When we request the smaller query (Source code extract 6.2) modified to contain also the size limits.

```
query users {
    users {
        id
        name
        comments {
            title
            body
        }
        posts {
            title
            body
            comments {
                title
                body
                author {
                    name
                }
            }
        }
        announcements {
            body
            summary
        }
        suggestions {
            body
            tag
        }
    }
}
```

Source code extract 6.2: Alternative 3 accepted test query.

Although it is a very slight difference, we can see a better response time due to fewer amounts of data being requested (Figure 6.7).

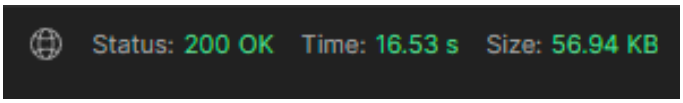

Figure 6.7: Alternative 3 accepted query time

For the final measurement evaluation we can see that:

- **Quality of the implementation**. The quality of implementations is the combination of the 2 alternatives, so no extra quality issue can be found.
- **Performance**. The performance of the queries is slightly better. This is due to the smaller amounts of data that we can request as said above.
- **Problematic Query rejection**. The capacity to reject queries is the same as alternative 2, so we do not have any improvement in this measurement.

As we can tell by the measurements alternative 3 has a slight advantage over alternative 2 and is considerably better than alternative 1. This evaluation impacts testing scenarios 2 and 3.

6.3 Analysis

To finalize we can see the results of each testing scenario (Table 6.1).

Table 6.1: Testing Scenarios.

Alternatives compared	**Result**
Alternatives 1 and 2	Alternative 2 performs better
Alternatives 1 and 3	Alternative 3 performs better
Alternatives 2 and 3	Alternative 3 performs better

Analysing the results of each testing scenario it is possible to say that the best alternative would be the implementation of alternative 3 which combines the two other alternatives leading to a very accurate query rejection and better performance due to the limiting approach. If needed, both smaller amounts and cost limits can be adjusted to get a more restricted API. Although this is the result in this prototype, to fully validate these alternatives they would need to be tested in many other prototypes to get more accurate results.

Chapter 7

Conclusion

This chapter summarizes the work done, its limitations and what can be improved in the future..

7.1 Summary

Three different approaches were explored to have a more secure and reliable GraphQL API, it can be said that all could achieve some sort of success.

The first alternative aimed at the quantity of data that could be requested by limiting both the number of elements and nested queries that could be requested. It is a straightforward approach and it is successful in what is designed for. It can be considered the minimum to be done to have a somewhat safe GraphQL API that cannot be easily taken down.

The second alternative focus on the further capability of rejecting expensive/problematic queries. The followed approach had a simple idea of defining values for each type and calculating what would be the value of the requested query and comparing it to the maximum value. This approach makes the API more robust and is an option to be considered for teams that implement GraphQL APIs.

The third and final alternative joins both worlds. By doing that the API gets the best from both alternatives joining the limiting capabilities with the capacity to reject problematic queries right at the starting point.

It could be understood from the obtained results, that any GraphQL API should at least implement a limiting approach since it provides an initial level of security without compromising both performance and effort. However, the use of Query cost analysis or the combination of both is recommended since it provides a more robust API without having any impact on performance according to the tests realized.

7.2 Objectives

The main objective of this project was to "explore current approaches that try to minimize these concerns and identify their advantages and disadvantages by developing solution(s) while acknowledging their impact on the system's performance". By looking into our objective, it can be justified that it was achieved. Multiple approaches were studied and from the ones studied, the more related to the problem were picked. In the approaches followed, both advantages and disadvantages were identified as well as possible improvements. In this specific implementation, the impact on the API performance was not found.

7.3 Limitations and Future Work

The main limitation found during the development of this project was the lack of time to explore more approaches, apply the chosen approaches to more than one project and, possibly implement alternatives to current know approaches.

For future work, it is fundamental to apply these same approaches in multiple projects to get more accurate results. The results obtained from those applications are to be compared with the current results.

Furthermore, a more extensive study on the current approaches and possible improvements should be suggested or implemented in the libraries used.

7.4 Final considerations

The work done in this project allowed the author to explore new technologies and important concepts related to the master's degree in software engineering, namely the security of the current API implementations and good practices in architecture design and implementation.

This work managed to provide an interesting challenge and created many new findings that can be useful in the future, both from an academic and professional perspective.

The implementation of safer strategies during API development as well as the use of GraphQL itself provided knowledge and skills that will have an immediate impact on the author's professional life as well as its organization.

Bibliography

4Catalyzer/graphql-validation-complexity: Query complexity validation for GraphQL.js (n.d.). url: https://github.com/4Catalyzer/graphql-validation-complexity.

Beck, Roman, Sven Weber, and Robert Wayne Gregory (Sept. 2013). "Theory-generating design science research". In: *Information Systems Frontiers* 15 (4), pp. 637–651. issn: 13873326. doi: 10.1007/S10796-012-9342-4.

Brito, Gleison and Marco Tulio Valente (Mar. 2020). "REST vs GraphQL: A controlled experiment". In: *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020*, pp. 81–91. doi: 10.1109/ICSA47634.2020.00016.

Brown, Simon (2022). *The C4 model for visualising software architecture*. url: https://c4model.com/.

Cha, Alan et al. (Nov. 2020). "A principled approach to GraphQL query cost analysis". In: *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 257–268. doi: 10.1145/3368089.3409670.

Cho, Joon Ho (June 2019). *joonhocho/graphql-scalar: Configurable custom GraphQL Scalars (string, number, date, etc) with sanitization / validation / transformation in TypeScript.* url: https://github.com/joonhocho/graphql-scalar.

Emmy, Irere (Sept. 2021). *Irere123/Board: Board is taking community communication to the stars*. url: https://github.com/Irere123/Board.

GitHub - apollographql/persistgraphql: A build tool for GraphQL projects. (2018). url: https://github.com/apollographql/persistgraphql.

GraphQL - OWASP Cheat Sheet Series (n.d.). url: https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html.

GraphQL | A query language for your API (2022). url: https://graphql.org/.

Lytek, Michał (Nov. 2020). *MichalLytek/type-graphql: Create GraphQL schema and resolvers with TypeScript, using classes and decorators!* url: https://github.com/MichalLytek/type-graphql.

Magazine, Security (Dec. 2021). *Researchers discover GraphQL authorization flaws in fintech SaaS platform | Security Magazine*. url: https://www.securitymagazine.com/articles/96657-researchers-discover-graphql-authorization-flaws-in-fintech-saas-platform.

Mahalakshmi, S (2021). *Value Analysis - Definition, Steps, Examples, How it Work?* url: https://www.wallstreetmojo.com/value-analysis/.

Mavroudeas, Georgios et al. (Aug. 2021). "Learning GraphQL Query Costs (Extended Version)". In: url: http://arxiv.org/abs/2108.11139.

Monteiro, Luis (Apr. 2022). *Commits · luisbrunosw/tmdei$_1$171038*. url: https://github.com/luisbrunosw/tmdei_1171038.

Muench, Justin (Feb. 2022). *Why Do We Need GraphQL?. And how can it speed up our apps | by Justin Muench | Better Programming*. url: https://betterprogramming.pub/why-do-we-need-graphql-43ea26d0efc4.

Orlivskyi, Serhii, Bohdan Deomin, and Olga Averianova (Aug. 2021). "Pagination and Its Efficient Methods for RESTful Web Services". In: *2021 IEEE 3rd Ukraine Conference on Electrical and Computer Engineering, UKRCON 2021 - Proceedings*, pp. 567–571. doi: 10.1109/UKRCON53503.2021.9575139.

pa-bru/graphql-cost-analysis: A Graphql query cost analyzer. (N.d.). url: https://github.com/pa-bru/graphql-cost-analysis.

Pello, R (Oct. 2018). *Design science research — a short summary | by Rauno Pello | Medium*. url: https://medium.com/@pello/design-science-research-a-summary-bb538a40f669.

Porcello, Eve and Alex Banks (2018). *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. 1st. O'Reilly Media, Inc. isbn: 1492030716.

Rate Limiting GraphQL APIs by Calculating Query Complexity — Development (June 2021). url: https://shopify.engineering/rate-limiting-graphql-apis-calculating-query-complexity.

Scaling Netflix's API via GraphQL Federation (2) | Netflix TechBlog (Dec. 2020). url: https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-2-bbe71aaec44a.

Shtatnov, A and R. S. Ranganathan (Dec. 2018). *Our learnings from adopting GraphQL | by Netflix Technology Blog | Netflix TechBlog*. url: https://netflixtechblog.com/our-learnings-from-adopting-graphql-f099de39ae5f.

slicknode/graphql-query-complexity: GraphQL query complexity analysis and validation for graphql-js (n.d.). url: https://github.com/slicknode/graphql-query-complexity.

Soliman, Mostafa and Marianne A. Azer (Feb. 2019). "Web application API blind denial of service attacks". In: *ICENCO 2018 - 14th International Computer Engineering Conference: Secure Smart Societies*, pp. 249–253. doi: 10.1109/ICENCO.2018.8636115.

Stoiber, M (Feb. 2018). *Securing Your GraphQL API from Malicious Queries - Apollo GraphQL Blog*. url: https://www.apollographql.com/blog/graphql/security/securing-your-graphql-api-from-malicious-queries/.

Stuart, M (2018). "GraphQL: A Success Story for PayPal Checkout | by Mark Stuart | The PayPal Technology Blog | Medium". In: url: https://medium.com/paypal-tech/graphql-a-success-story-for-paypal-checkout-3482f724fb53.

Suriadi, Suriadi, Andrew Clark, and Desmond Schmidt (2010). "Validating denial of service vulnerabilities in web services". In: *Proceedings - 2010 4th International Conference on Network and System Security, NSS 2010*, pp. 175–182. doi: 10.1109/NSS.2010.41.

Appendix A

Domain concepts

A.1 Initial domain

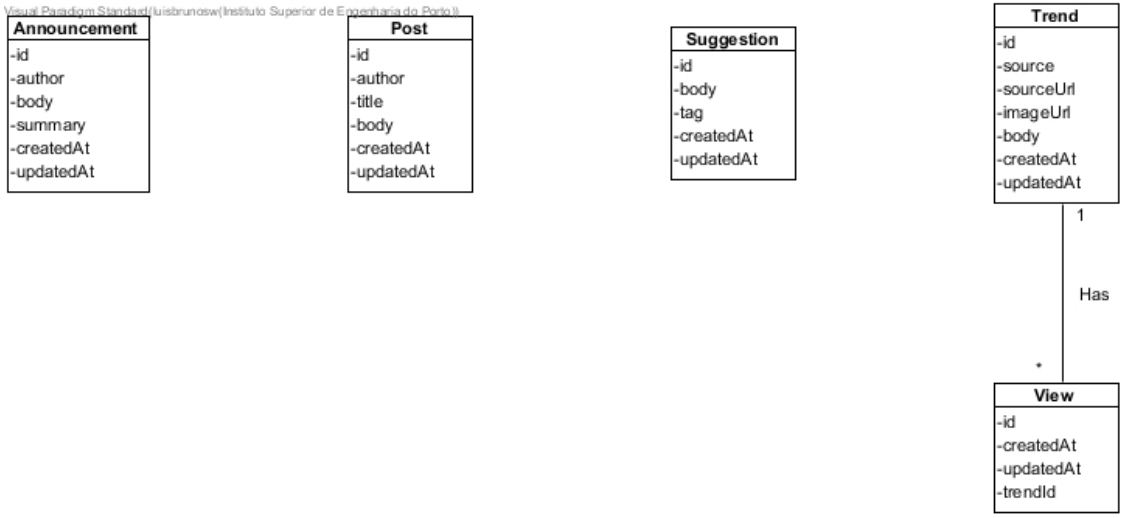

Figure A.1: Initial domain

A.2 Improved domain

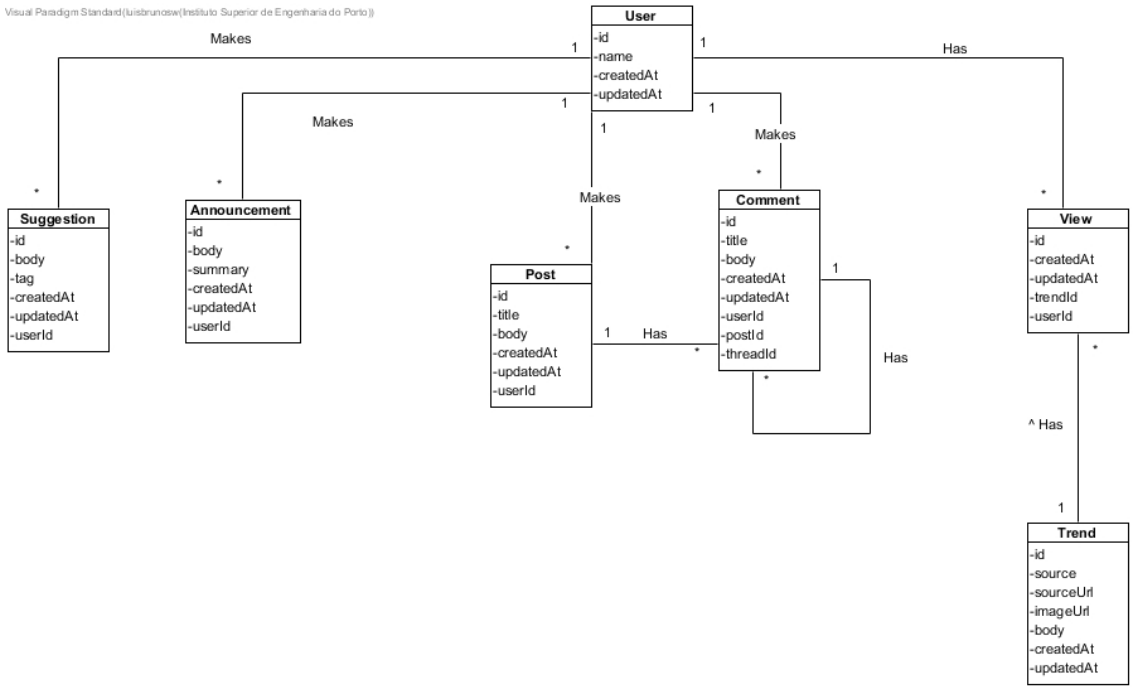

Figure A.2: Improved domain

Appendix B

Architectural Design

B.1 System Context Diagram

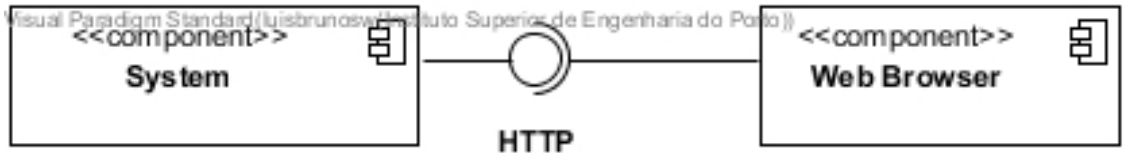

Figure B.1: System Context Diagram

B.2 Container Diagram

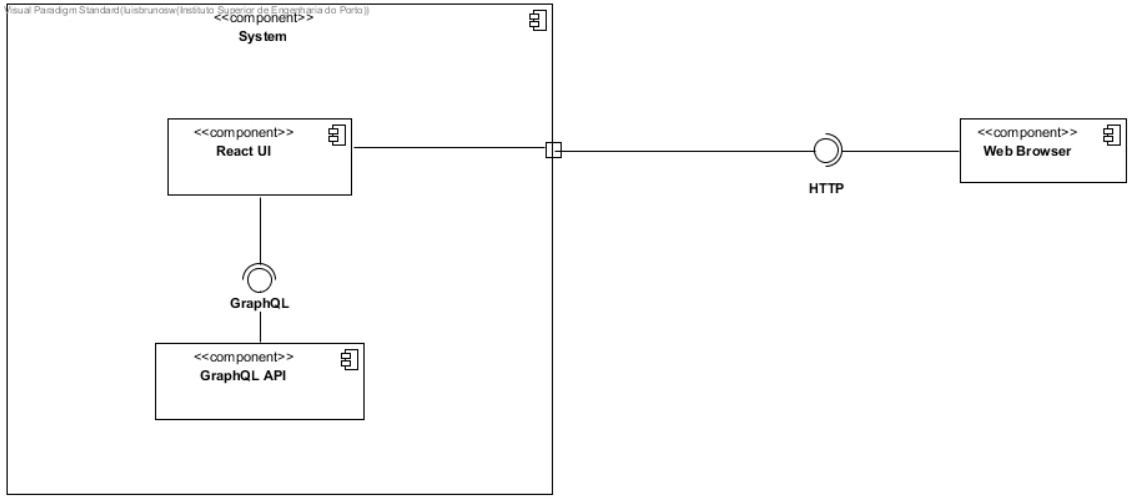

Figure B.2: Container Diagram

B.3 Component Diagram API

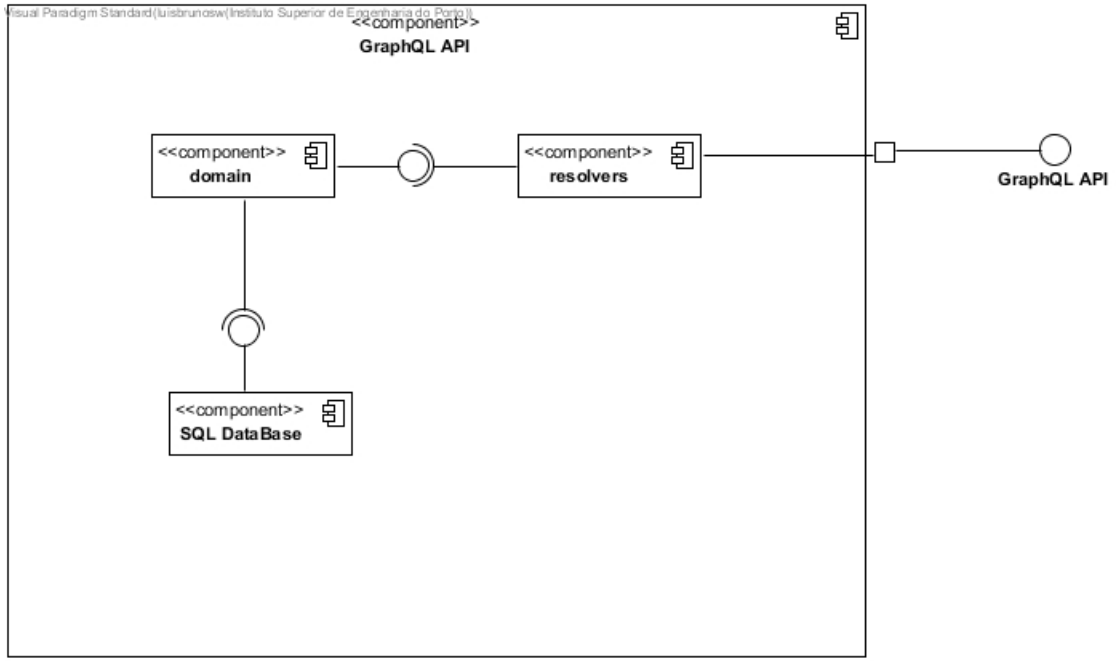

Figure B.3: Component Diagram API

B.4 Component Diagram UI

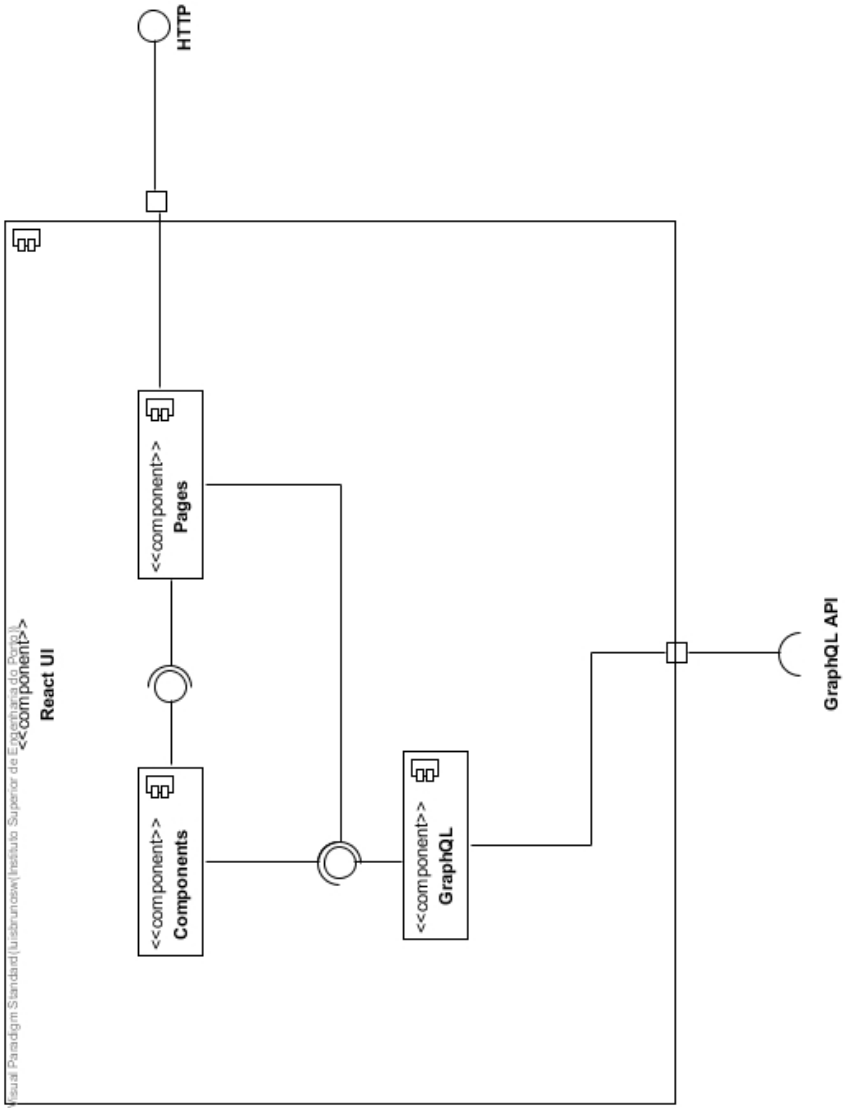

Figure B.4: Component Diagram UI

Appendix C

Functional Requirements Design

C.1 Use Case Diagram

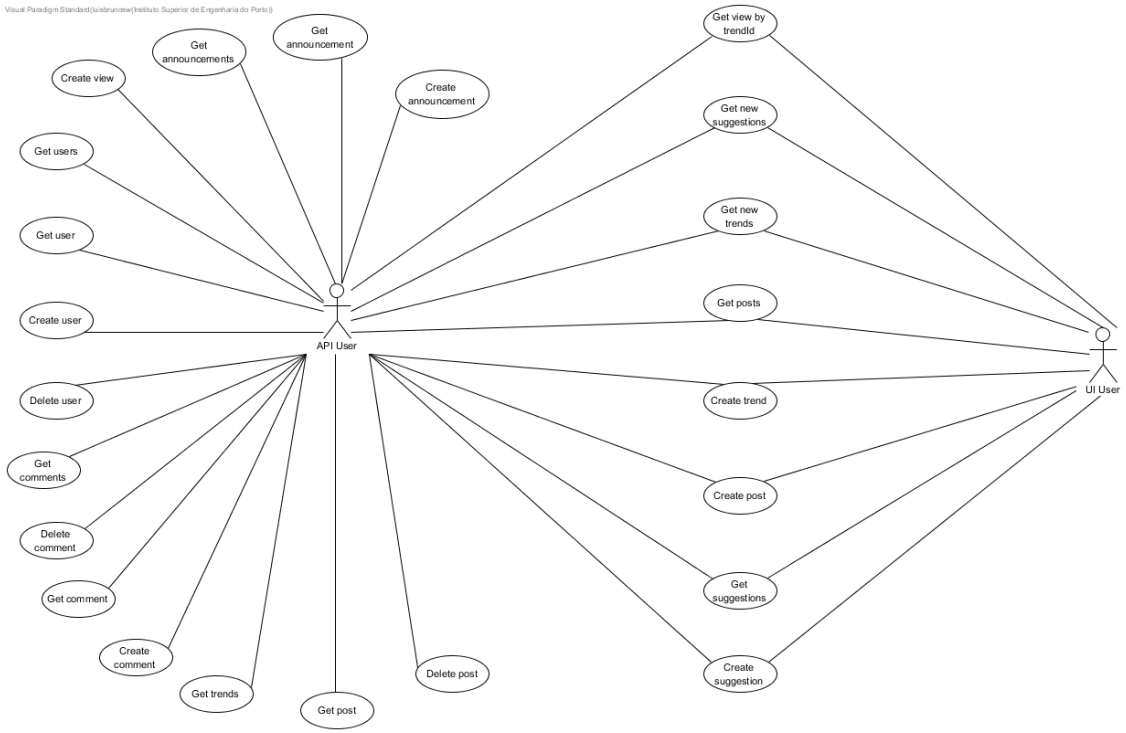

Figure C.1: Use Case Diagram

C.2 FR1 Sequence Diagram

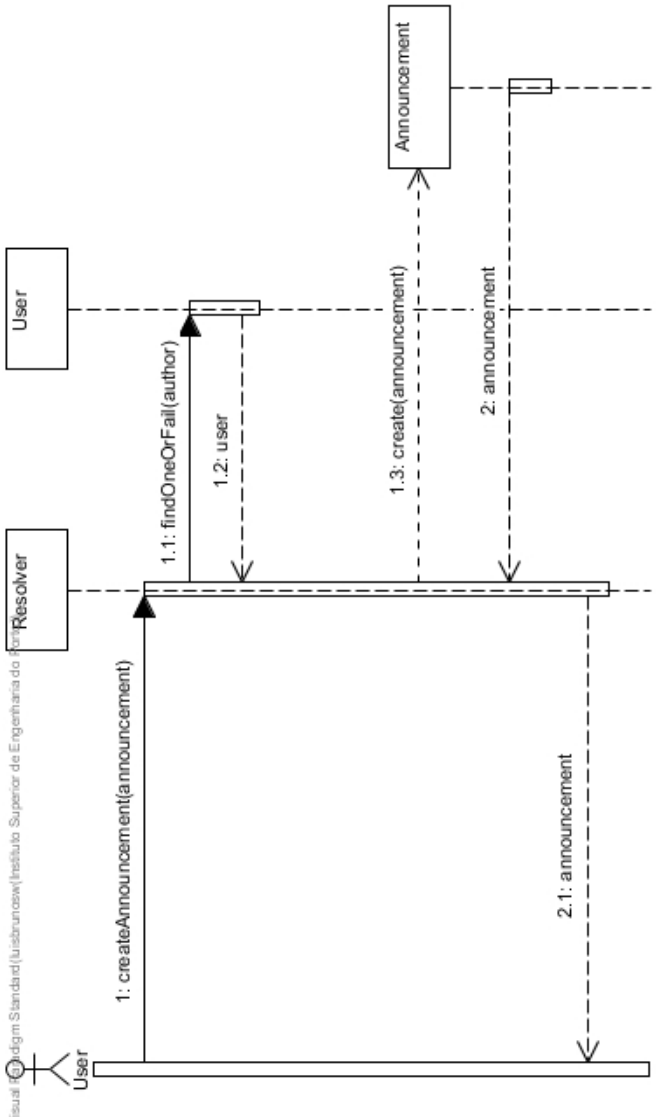

Figure C.2: FR1 Sequence Diagram

C.3 FR10 Sequence Diagram

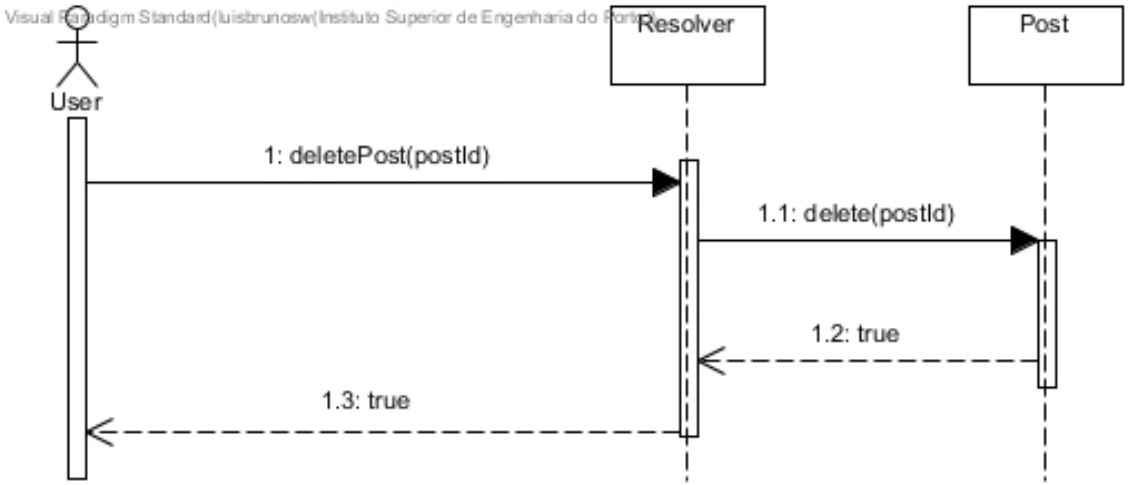

Figure C.3: FR10 Sequence Diagram

C.4 FR19 Sequence Diagram

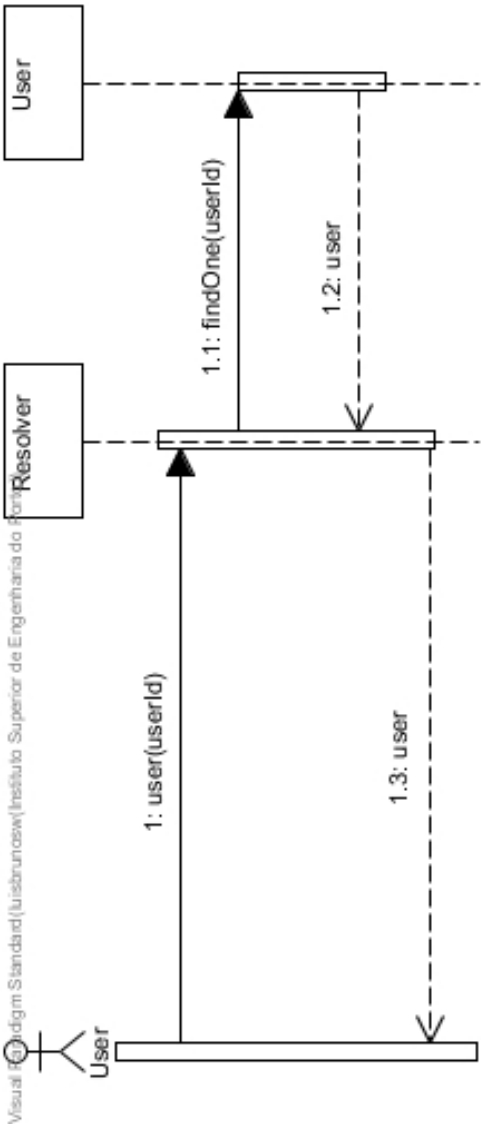

Figure C.4: FR19 Sequence Diagram

C.5 FR22 Sequence Diagram

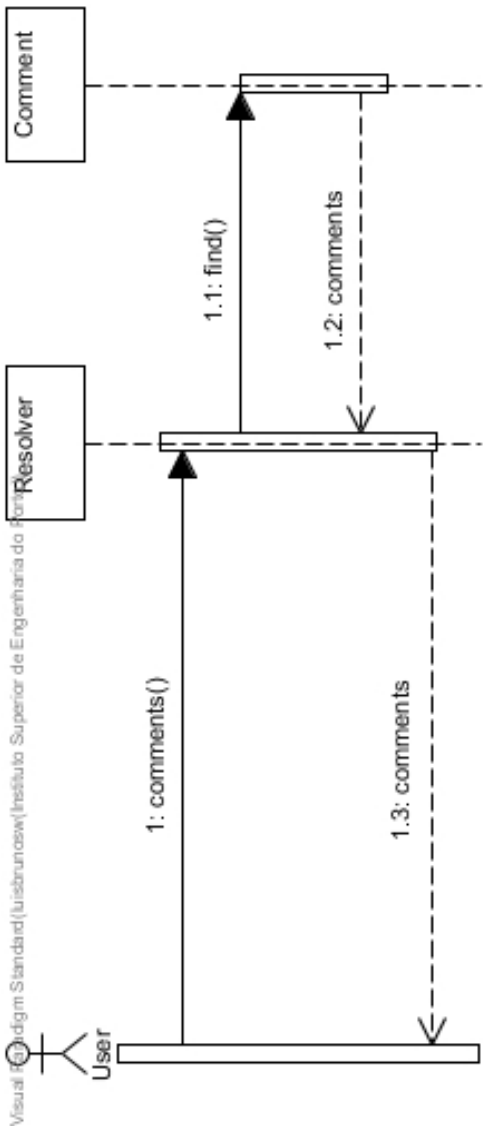

Figure C.5: FR22 Sequence Diagram

Appendix D

Alternative 3 Schema

```
type Query {
  posts(filter: ListFilter!): [Post!]!
  post(postId: String!): Post!
  comments(filter: ListFilter!): [Comment!]!
  comment(commentId: String!): Comment!
  users(filter: ListFilter!): [User!]!
  user(userId: String!): User!
  announcements(filter: ListFilter!): [Announcement!]!
  announcement(annId: String!): Announcement!
  suggestions(filter: ListFilter!): [Suggestion!]!
  newSuggestions: [Suggestion!]!
  hello: String!
  trends(filter: ListFilter!): [Trend!]!
  newTrends: [Trend!]!
  views(trendId: String!): String!
}

type Post {
  id: ID!
  title: String!
  body: String!
  comments(filter: ListFilter!): [Comment]!
  createdAt: DateTime!
  updatedAt: DateTime!
  author: User!
}

type Comment {
  id: ID!
  title: String!
  body: String!
  answers(filter: ListFilter!): [Comment]!
  createdAt: DateTime!
  updatedAt: DateTime!
  thread: Comment!
  author: User!
  post: Post!
}
```

Source code extract D.1: Alternative 3 Schema.

```
input ListFilter {
  first: PaginationAmount!
}
scalar PaginationAmount
scalar DateTime

type User {
  id: ID!
  name: String
  comments(filter: ListFilter!): [Comment]!
  posts(filter: ListFilter!): [Post]!
  views(filter: ListFilter!): [View]!
  announcements(filter: ListFilter!): [Announcement]!
  suggestions(filter: ListFilter!): [Suggestion]!
  createdAt: DateTime!
  updatedAt: DateTime!
}

type View {
  id: ID!
  createdAt: DateTime!
  updatedAt: DateTime!
  author: User!
  trend: Trend!
}

type Trend {
  id: ID!
  source: String
  sourceUrl: String
  imageUrl: String
  views: [View]!
  body: String!
  createdAt: DateTime!
  updatedAt: DateTime!
}

type Announcement {
  id: ID!
  body: String!
  summary: String!
  createdAt: DateTime!
  updatedAt: DateTime!
  author: User!
}
```

Source code extract D.2: Alternative 3 Schema.

```
type Suggestion {
  id: ID!
  body: String!
  tag: String!
  createdAt: DateTime!
  updatedAt: DateTime!
  author: User!
}

type Mutation {
  createPost(input: CreatePostInput!): CreatePostResponse!
  deletePost(postId: String!): Boolean!
  createComment(input: CreateCommentInput!):
   CreateCommentResponse!
  deleteComment(commentId: String!): Boolean!
  createUser(input: CreateUserInput!): CreateUserResponse!
  deleteUser(userId: String!): Boolean!
  createAnnouncement(
    input: CreateAnnouncementInput!
  ): CreateAnnouncementResponse!
  createSuggugestion(input: CreateSuggestionInput!):
   CreateSuggestionResponse!
  createTrend(input: CreateTrendInput!): CreateTrendResponse!
  createView(input: CreateViewInput!): CreateViewResponse!
}

type CreatePostResponse {
  ok: Boolean!
  post: Post!
}

input CreatePostInput {
  author: String!
  title: String!
  body: String!
}

type CreateCommentResponse {
  ok: Boolean!
  comment: Comment!
}

input CreateCommentInput {
  author: String!
  post: String
  title: String!
  body: String!
  thread: String
}
```

Source code extract D.3: Alternative 3 Schema.

```
type CreateUserResponse {
  ok: Boolean!
  user: User!
}

input CreateUserInput {
  name: String!
}

type CreateAnnouncementResponse {
  ok: Boolean!
  announcement: Announcement!
}

input CreateAnnouncementInput {
  author: String!
  body: String!
  summary: String!
}

type CreateSuggestionResponse {
  ok: Boolean!
  suggestion: Suggestion!
}

input CreateSuggestionInput {
  body: String!
  tag: String!
  author: String!
}

type CreateTrendResponse {
  ok: Boolean!
  trend: Trend!
}

input CreateTrendInput {
  source: String!
  sourceUrl: String
  imageUrl: String
  body: String!
}

type CreateViewResponse {
  ok: Boolean!
  view: View!
}
```

Source code extract D.4: Alternative 3 Schema.

```
input CreateViewInput {
  trend: String!
  author: String!
}
```

Source code extract D.5: Alternative 3 Schema.

Appendix E

Alternative 3 Folder Structure

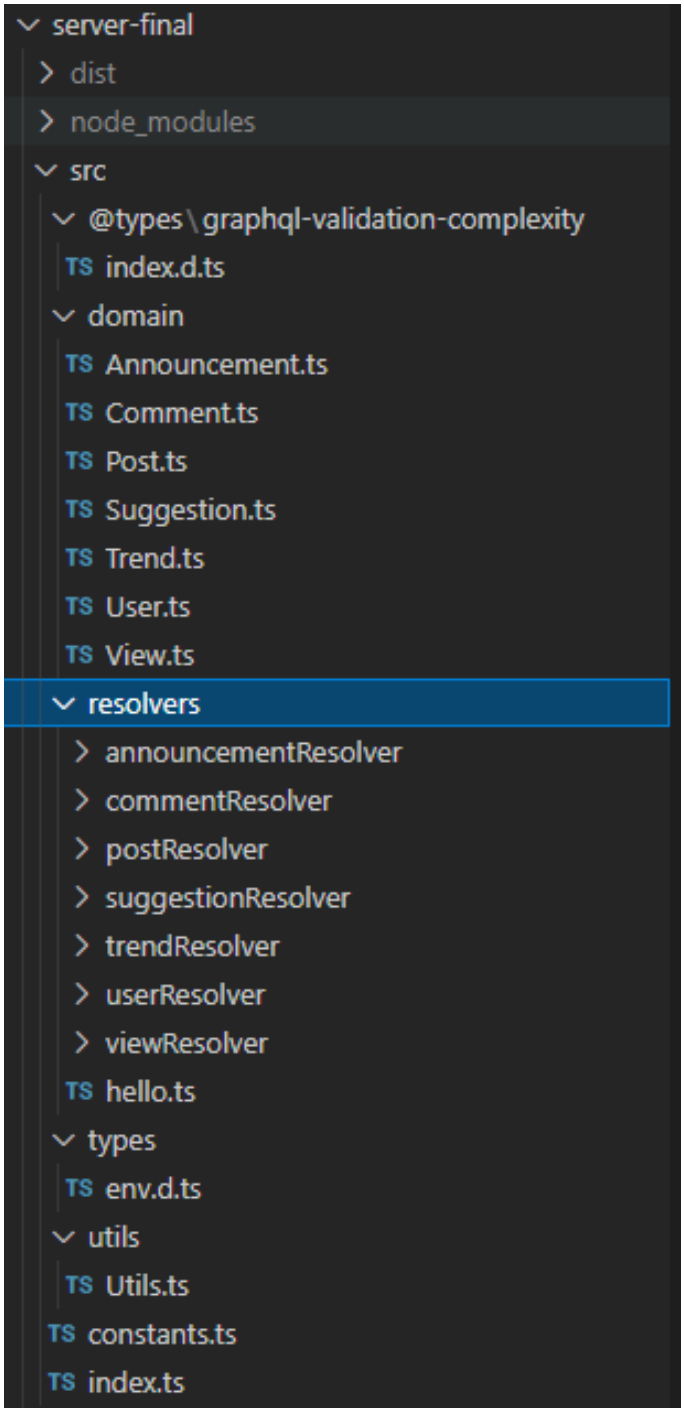

Figure E.1: Alternative 3 Folder Structure