INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Analysis of GraphQL performance: a case study

MAFALDA ISABEL FERREIRA LANDEIRO
Julho de 2019

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Analysis of GraphQL performance: a case study

Mafalda Isabel Ferreira Landeiro

Dissertation to obtain a Master's degree in Informatics Engineering, Area of Specialization in Software Engineering

Academic supervisor: Isabel de Fátima Silva Azevedo

Enterprise supervisor: José Carlos Sousa

Porto, July 2019

However, difficult life may seem,
there is always something you can do and succeed at.
It matters that you do not just give up.
(Stephen Hawking)

Dedication

To my family 

To my boyfriend 

To Bia 

Resumo

Atualmente os aplicativos da Web têm um papel relevante, com um grande número de aparelhos conectados à Internet e os dados são transmitidos entre plataformas distintas a um ritmo sem precedentes. Vários sistemas e plataformas de tipos diferentes, como web e móveis, exigem que os aplicativos se adaptem de maneira rápida e eficiente às necessidades dos consumidores.

Em 2000, o Representation State Transfer (REST) foi apresentado e foi rapidamente adotado pelos desenvolvedores. No entanto, devido ao crescimento dos consumidores e às necessidades distintas, este estilo arquitetónico, na forma como é utilizado, revelou algumas fragilidades relacionadas com o desempenho e flexibilidade das aplicações. Estas são ou podem ser endereçadas com GraphQL. Apesar de ser uma tecnologia recente, já é usada por grandes empresas como Facebook, Netflix, GitHub e PayPal.

Recentemente, uma plataforma do INESC TEC, denominada IRIS, enfrentou os mesmos problemas de desempenho e a possibilidade de adoção do GraphQL foi considerada. Várias alternativas com GraphQL foram estudadas e analisadas de forma a verificar se poderiam beneficiar o IRIS em termos de desempenho e flexbilidade.

Uma das conclusões deste estudo é que todas as alternativas testadas revelam, no geral, melhores resultados de desempenho, tendo em consideração o tempo de resposta e o tamanho da resposta. No entanto, a utilização de uma alternativa constituída apenas por GraphQL apresenta-se como a melhor solução para melhorar o desempenho e flexibilidade de uma aplicação.

Palavras-chave: GraphQL, REST, Webservice, Performance

Abstract

Web applications today play a significant role, with a large number of devices connected to the Internet, and data is transmitted across disparate platforms at an unprecedented rate. Many systems and platforms of different types, such as web and mobile, require applications to adapt quickly and efficiently to the needs of consumers.

In 2000, the Representation State Transfer (REST) was introduced, and the developers quickly adopted it. However, due to the growth of consumers and the different needs, this architectural style, in the way it is used, revealed some weaknesses related to the performance and flexibility of the applications. These are or can be addressed with GraphQL. Despite being a recent technology, it is already used by big companies like Facebook, Netflix, GitHub, and PayPal.

Recently, an INESC TEC platform, called IRIS, faced the same performance problems and the possibility of adopting GraphQL was considered. Several alternatives with GraphQL were studied and analyzed to see if they could benefit IRIS in terms of performance and flexibility.

One of the conclusions of this study is that all of the alternatives tested show, overall, better performance results, taking into account response time and response size. However, the use of an alternative consisting solely of GraphQL is the best solution to improve the performance and flexibility of an application.

Keywords: GraphQL, REST, Webservice, Performance

Acknowledgment

This thesis would not have been possible without the cooperation and goodwill of those directly and indirectly involved in the project. I want to express my sincere thanks to all.

To ISEP, where all my academic path was made, and that transmitted me all the necessary knowledge for being capable of concluding a project like this.

To INESC TEC, to allow me to conjugate my thesis with my daily job and to give me the possibility to dedicate some time to this project.

To my academic supervisor, Professor Isabel Azevedo, for being not only a great supervisor, guiding my work and debating some ideas, but also for giving me such positive energy and motivation, during all the thesis. Thank you for all the patience and availability. After two years, we made it!

To my enterprise supervisor, José Carlos Sousa, for backing me up during this journey and supporting me in all the decisions made.

To the executive director of INESC TEC, Professor Gabriel David, for providing me the time and space need to dedicate to this thesis and for giving me some guidance about possible alternatives to explore.

To my colleagues in INESC TEC, specially to: Jaime Dias, for all the time helping me and the openness to discuss some ideas; to Fábio Alves, for sharing opinions about this document; to Soraia Ramos, it was enjoyable that we were doing the thesis at the same time and had shared so much, even though we are from distinct areas.

To my family, for keeping on supporting me in all the moments and believing in my abilities. Thank you for being always there for me.

To my boyfriend, the most significant support of all this journey. Thank you for all the patience and for believing in me all the time. You were the essential key of all my positive mind and strength to finish this project.

To Bia, my cat, because there is no better company for writing than her. Thank you for lending me a paw!

Table of Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms and Symbols

AHP	Analytic Hierarchy Process
ANOVA	Analysis of Variance
API	Application
CRUD	Create, Read, Update and Delete
DI	Developer Interface
DX	Developer Experience
FEI	Front End of Innovation
HATEOAS	Hypermedia As The Engine Of Application State
HSD	Honestly Significantly Different
HTTP	Hypertext Transfer Protocol
INESC TEC	Institute of Systems and Computer Engineering, Technology and Science
IRIS	INESC TEC Research Information System
ISEP	Porto School of Engineering
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
NCD	New Concept Development
POM	Project Object Model
R&D	Research and Development
R&DT	Research and Technology Development
REST	Representational State Transfer
SDL	Schema Definition Language
SQL	Structured Query Language

UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language
YAML	YAML Ain’t Markup Language

1 Introduction

What if I fall?
Oh, but my darling, what if you fly?
(Erin Hanson)

This chapter presents the project developed during the master's degree in Informatics Engineering, area of specialization in Software Engineering, at Porto School of Engineering (ISEP). Firstly, there is a brief contextualization and characterization of the problem studied. Then, the primary goals are introduced, and the applied methodologies. Lastly, it describes the structure of the document.

1.1 Context

In the last few years, there has been a growth of Web Applications usage and increasing necessity of obtaining fast responses flexibly, to give to the application consumers the best experience possible. Architectures that include services as data management/producers are usually because of the need to centralize the core of applications and to interconnect with multiple systems and a variety of platforms types.

Once introduced, Representational State Transfer (REST) was quickly adopted by developers, because of its main features: scalability, interoperability, simplicity, and extensibility (Fielding and Taylor, 2000). However, on big organizations that have to deal with complex entities and requests, this architectural style started to reveal some drawbacks.

In 2015, Facebook presented GraphQL, a query language that can improve flexibility, performance, and memory use of applications (APIs) (Porcello and Banks, 2018). Leaving behind the necessity of multiple endpoints calls and defining which data the client wants to receive, quickly gain some fans, and it started to question the possibility of this be the end of REST.

Some studies were made comparing the two technologies using different criteria. Some similarities between them were pointed out, as well as the lack of maturity of GraphQL (Stubailo, 2017). Guillen-Drija analyzed some quality attributes and concluded that, in atomic calls, REST presents better performance, while GraphQL deals better with overfetching and underfetching (Guillen-Drija et al., 2018). Vázquez-Ingelmo described best networking response times with GraphQL (Vázquez-Ingelmo et al., 2017).

Other companies, like Netflix (Shtatnov and Ranganathan, 2018), GitHub (Torikian et al., 2016), and Paypal (Stuart, 2018), have also started to adopt GraphQL as an alternative to REST APIs, and they presented their conclusions highlighting the benefits that technology brought to them.

1.2 Problem

At Institute of Systems and Computer Engineering, Technology and Science (INESC TEC), there is a web platform called INESC TEC Research Information System (IRIS) with one year, that allows managing information, such as research and human resources data, from the organization.

In this application, there are different levels of complexity in searching for information, mainly on the project management area, and the actual solution presents some performance issues. It is known that if the response time is longer than ten seconds, users will lose their attention (Möller, 2010). This web application has a backend that follows the REST architecture, and the frontend is one of the primary consumers from the REST APIs available.

A lot of new features have been added to IRIS, and the necessity to integrate with distinct platforms has also been growing, especially for the project management module. The use of REST APIs started to present some issues related to data querying, such as:

- The necessity to call more than one endpoint to obtain the necessary data;
- Each consumer has its necessities, and it is receiving more than the necessary data;
- Performance issues also affected by the previous points.

Some temporary solutions were considered, like adapting the server response to supply most of the data required in only one request. However, that brought an increase in the response size and only a slight improvement in performance.

Besides, some studies prove that REST APIs have been used to answer the necessities of distinct consumers, but even following the best practices, this solution does not present the ideal elasticity, since in some cases can require a lot of endpoints requests or receive additional data (Vázquez-Ingelmo et al., 2017). Thus, a more flexible solution with better performance is aimed.

Although the recent developments with GraphQL, there is a lack of understanding about the possibility to conjugate its flexibility with performance.

1.3 Objectives

To improve the flexibility of applications, the use of GraphQL can bring some benefits (Nordic APIs, 2017). GraphQL is a query language developed by Facebook, that allows to query and manipulate data (Facebook Inc., 2018), but also a specification and a set of tools.

Although, there are some wrong ideas that this language is a replacement to REST, comparing each other it is possible to find out that they can work well together, empowering data searches (Sturgeon, 2017), but the evidence is scarce.

So, the main goal of this thesis is to explore the use of GraphQL for data querying to improve both the performance and flexibility of the API. A solution is to be developed and assessed. The analysis of the introduction of the mentioned technology in search of data will mitigate the uncertainty related to the following research question:

- Can GraphQL improve searching performance, while also providing flexibility, when compared to the existing solution?

Exploring the issue may provide a better solution to apply to IRIS and similar software applications or leave more outlined.

1.4 Methodology

For achieving the objectives (section 1.3), the following methodological phases were followed:

- **Problem interpretation**: improve the understanding of the current problems and possible and desirable improvements, not only to the web application in use for testing but also reported in the literature;

- **State of the art**: obtain the information need about the subjects in the study: REST and GraphQL and analyze some of the work done by researchers and conclusions reached by real cases that tried the transition from REST to GraphQL. Since GraphQL is a recent technology, there are only a few scientific papers related to the subject of this thesis so that it will be used grey literature;

- **Value analyses**: study the value of this project and define it;

- **Explore technologies**: considering the alternatives for improving data request performance, explore the best practices to apply, using GraphQL;

- **Design**: define requirements to implement on the two prototypes that will be used for testing (GraphQL standalone solution and GraphQL with REST solution);

- **Implementation**: taking into consideration the design, implement the two distinct approaches in a coherent solution;

- **Test and experimentation**: establish the criteria of experimentation and apply it on tests and test the features implemented on each solution;

- **Result analysis**: compare the results obtain by the new solutions and compare with the results of the previous implementation, to highlight potential applications and to identify limitations.

1.5 Organization

Here a brief outline of the content of each of the X chapters:

- **Introduction:** It gives a general presentation of the subject studied for this thesis. It is made a contextualization and description of the problem that motivates this project, followed by a definition of the objectives and the methodology used to achieve them. Finally, it is described the structure of the document;

- **Context:** the information about the institution and application where the solution of this thesis will be applied is shown in this chapter. There is also a more in-depth presentation of the problem and the definition of the project value, including a detailed analyzation of the product value, with focus on the business and innovation process, the value offer, the value proposition, and the canvas business model;

- **State of the Art:** in this chapter, some essential knowledge about REST and GraphQL is presented. It also contains a summary of three studies that compares them and the conclusions that they achieved. After that, there is a brief description of GraphQL adoption by three companies that early used REST (Netflix, GitHub, and PayPal) and the highlights of that process. At the end there is a short presentation of some of the remaining technologies that were also used;

- **Design:** This chapter includes a requirements analysis to define the architecture and design of the solution. In this chapter exists detail about the system actors, functional requirements, and other requirements necessary for the project. It also contains the design of the prototypes that are going to be compared. All the architectural decisions and justifications can be found in this chapter;

- **Implementation:** the implementation of the prototypes is described in this section, with the UML diagrams that represent them and mention the issues found during their development;

- **Tests and Solution Evaluation:** this chapter has expressed the tests of each solution, and it is presented what and how they are going to be validated. There is also a subsection for comparing results;

- **Conclusion:** the last chapter provides the conclusions about the describe work in this document, mentions some of the difficulties found, what can be done in the future and the contribution this thesis gave to the specific case of IRIS.

At the end of the document, some appendixes can provide more detail information about some work done during this project, namely:

- Appendix A – GraphQL Schema;

- Appendix B – Prototype 1 POM;

- Appendix C – Prototype 2 POM;

- Appendix D – Acceptance tests;

- Appendix E – Example of a sensor configuration;

- Appendix F – Descriptive statistics (quantitative data) for time;

- Appendix G – Descriptive statistics (quantitative data) for size.

2 Context

You cannot go back and change the beginning,
but you can start where you are and change the ending.
(C. S. Lewis)

The chapter aims to provide some knowledge about the institution involved in this thesis, which allows to contextualize the problem and present it with better detail. In the end, it makes the value analysis of the project solution.

2.1 INESC TEC

The Institute of Systems and Computer Engineering, Technology and Science (INESC TEC) is a private institution, categorized as an interface between the academic and business worlds (INESC TEC, 2017), that focus on four areas:

- Scientific research and technological development;
- Technology transfer;
- Advanced consulting and training;
- Pre-incubation of new technology-based companies.

It has thirteen Research and Development (R&D) Centres sorted on four domains: Computer Science, Industry and Innovation, Networked Intelligent Systems and Power and Energy, and it

present a continuous growing on activities in Research and Technology Development (R&TD) programs (Figure 1).

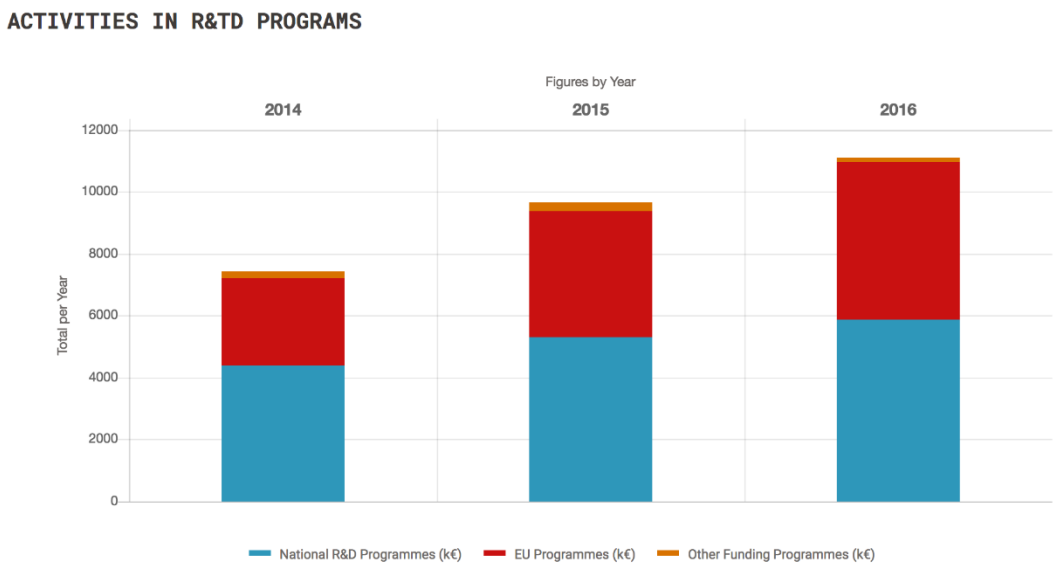


Figure 1 - Activities in R&TD programs of INESC TEC, from (INESC TEC, 2017)

The growth of activities led to the creation of a new web platform, which is IRIS (Figure 2), to manage scientific data, but also other types of data that indirectly influence the scientific activities like the researchers' personal information.

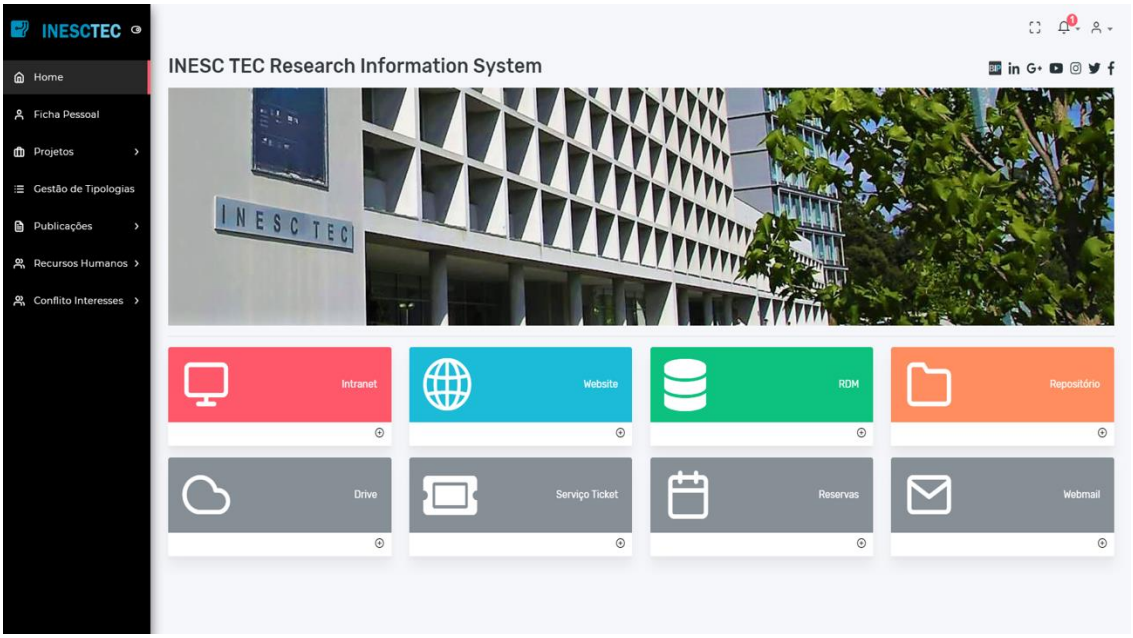


Figure 2 - IRIS home page

This platform is available to all the integrated researchers of the institution (about 842 researchers) and non-integrated with exclusive access (about 253 researchers), so currently 1095 users can access IRIS. Since November 2017 till June of 2019, there are 8155 distinct sessions created. Between November and December 2017, 923 sessions were created and comparing the first six months of 2018 with the first six months of 2019 (Figure 3), there is a growth of sessions. In January 2018, there is a vast number of sessions, because IRIS was being presented to the researchers, but the monthly sessions of 2018 would be between 250-300. At the beginning of 2019, the mean was still, but with the conclusion of the project management module, the number of sessions increased.

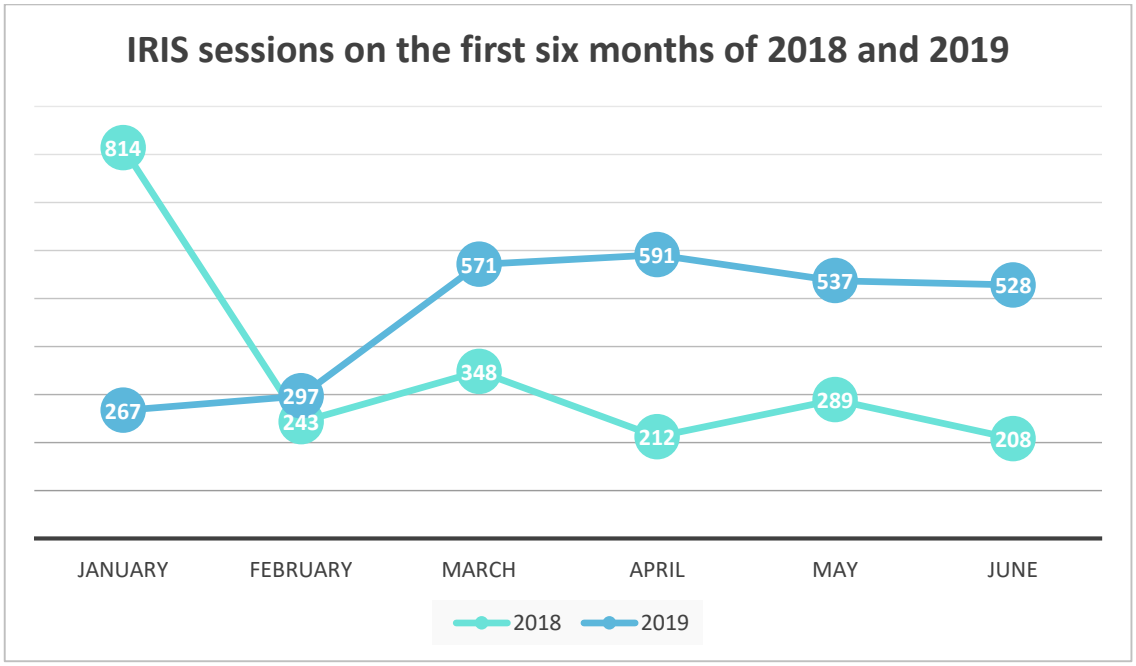


Figure 3 - IRIS sessions on the first six months of 2018 and 2019

IRIS is a relatively new solution that was made available at the end of 2017, with the possibility of managing and consulting data about the researchers, their projects, and their publications, but some problems have started to become evident. A lot of new features has been added, with a growing necessity to integrate with distinct platforms, as shown in Figure 4. IRIS has been assuming an essential role as an integrative platform.

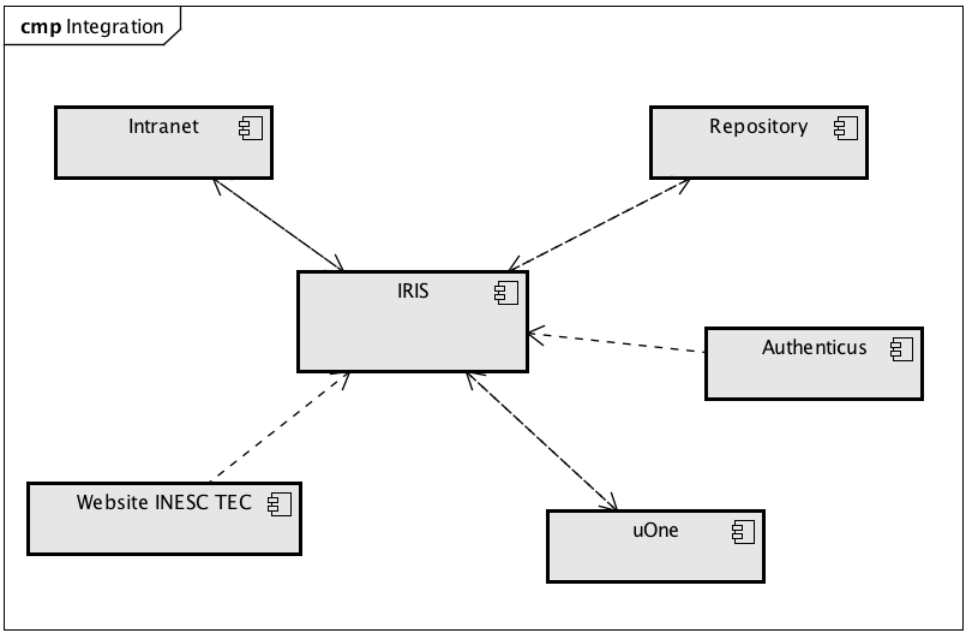


Figure 4 - IRIS integration with different platforms

The architecture used on IRIS (Figure 5) has a backend that uses REST APIs, implemented with Spring, and a frontend that consumes them, developed using Angular (User Interface) and it has a module of authorization that is also a REST API (Spring).

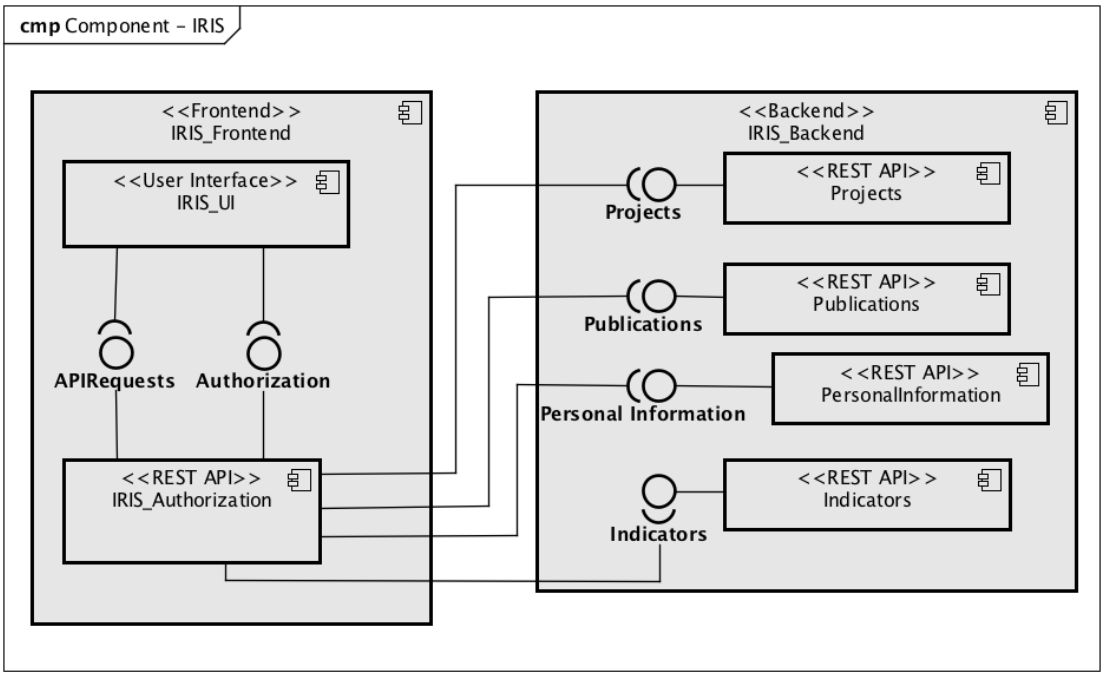


Figure 5 - Component diagram of IRIS

2.2 Problem

One of the latest modules of the application presented on the previous section is the project management area that involves a variety of users' profiles like project controllers, human resources technicians, projects managers, research units' coordinators, project team members, and others. This module also integrates with:

- **uOne:** is a platform that aids on daily project tasks management, in order to help the management of the teamwork;
- **Intranet:** provides the information about the project proposal;
- **Website:** consumes information about the projects to show to the public;
- **Repository:** stores all the documentation of the project.

On IRIS, there are different levels of complexity in data queries, mainly in this area (Figure 6). The use of REST APIs started to present some issues like the necessity to call more than one endpoint to obtain the necessary data, the fact that each consumer has its necessities and it is receiving excess data and the effect on the performance of the process.

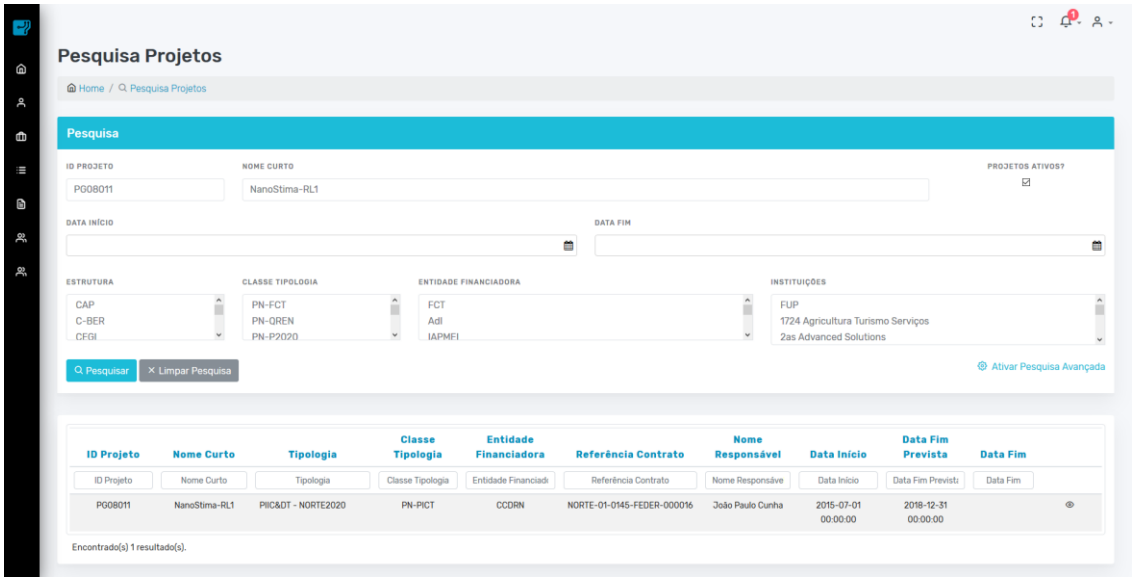


Figure 6 - Searching projects on IRIS

It is necessary to call six distinct endpoints for the search represented in Figure 6, namely, to obtain the table result presented in Figure 6:

- **/projectAPI/getProjectByOIBas**e: returns the information about project ID, short name, contract reference, begin date, preview end date, and end date, based on the project ID defined on search;

- **/projectAPI/getTypologyById:** returns the information about typology, based on the typology ID provided by the endpoint before;
- **/projectAPI/getClassTypologyById:** returns the information about the class of typology, based on the class typology ID provided by the typology endpoint;
- **/projectAPI/getFinancingEntityById:** return the information about the financing entity, based on the financing entity ID provided by the typology endpoint;
- **/projectAPI/getTeamByProjectId:** return the information about the team of the project, based on the project ID defined on search;
- **/personalInformation/getCollaboratorInformationByIdRH:** return the information of the project coordinator, based on the idRH (collaborator internal number) provided by the team endpoint;

The actual solution presents some performance issues like it is possible to notice with the case presented in Figure 7. Using the most simple search of a project, the sum of the response time is bigger than ten seconds, a value that is associated with loss of attention (Möller, 2010).

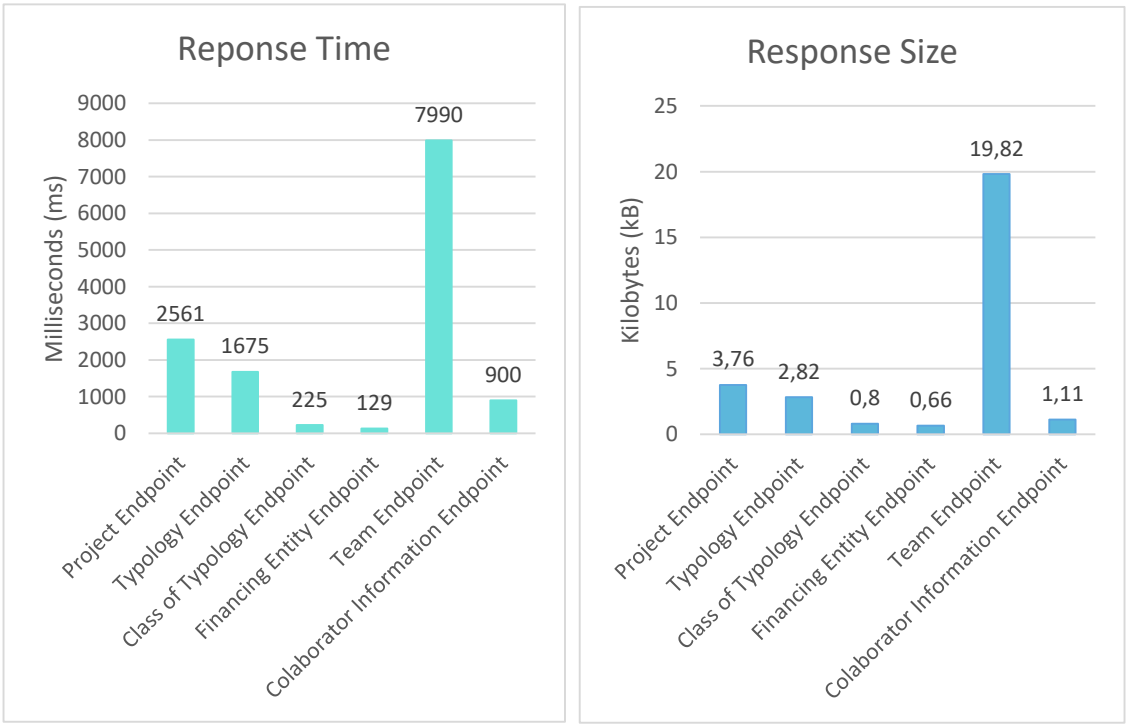


Figure 7 - Response time and size of a simple search on IRIS

Some temporary solutions were taking into consideration, like adapting the server response to supply most of the data required in only one request. However, that brought an increase of the response size and only an improvement of performance, like it is possible to see on the results

present on Figure 8, where it was made the same simple search reference before, and it makes only one call to the projects endpoint.

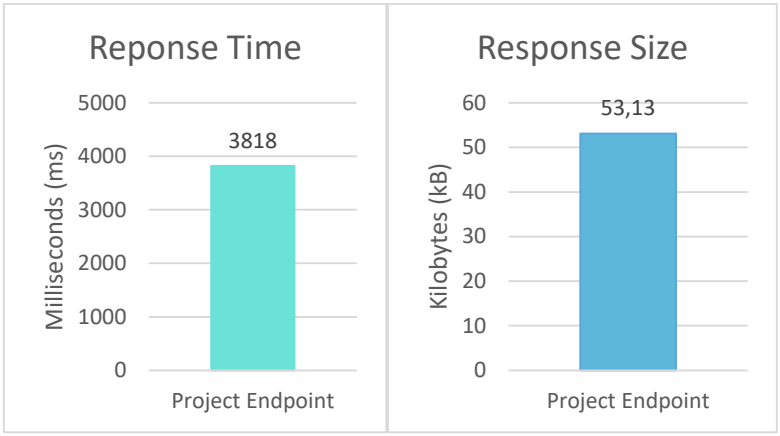


Figure 8 - Response time and size of search on IRIS after defining responses

Also, some studies show that REST APIs have been used to answer the necessities of distinct consumers, but even following the best practices, this solution does not present the ideal elasticity, since in some cases can require a lot of endpoints requests or receive unnecessary data (Vázquez-Ingelmo et al., 2017). Thus, a more flexible solution with better performance is aimed.

2.3 Value analysis

The value analysis is a process of analysis and evaluation of a product or a service that leads to an increase in its value with the lowest cost possible but keeping the quality.

In this subsection, the business and innovation process is presented, using the New Concept Development (NCD) Model. Then, the value offer and proposition, and the canvas business model of this project are described.

2.3.1 Business and Innovation Process

The NCD Model grants a common language and the characterization of the key elements that allow defining the Front End of Innovation (FEI) of a project/product. The NCD model (Figure 9) is composed of three areas:

- **Engine:** accounts for the vision, strategy, and culture by which the FEI is driven;
- **Five elements keys:** the opportunity identification, opportunity analysis, idea generation, and enrichment, idea selection and concept definition;
- External environmental factors.

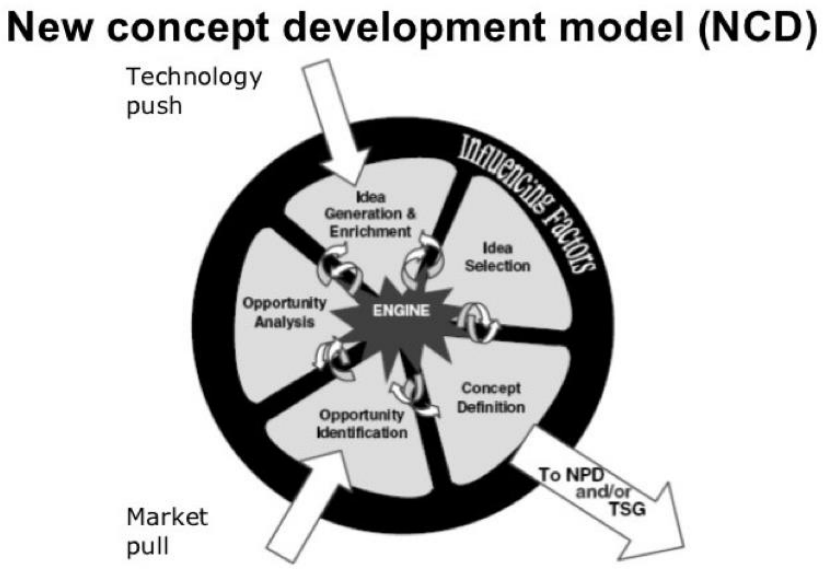


Figure 9 - New Concept Development Model, from (Vacek, 2012)

Using Peter Koen's model, this project has the following five keys elements:

- Opportunity identification

  With the creation of new features using REST APIs on IRIS, which will be continuously evolving, and the necessity of integration with distinct applications, there is a need to guarantee that the consumer of data presents the best performance and it is flexible enough to satisfy each consumer.

  Some methods/techniques/tools that can be used to analyze this essential element can be the technology trend analysis, which helps to collect data about which direction the company is going technically, and the road mapping, that can aid to find opportunities in the strategy adopted.

- Opportunity analysis

  For this project, the opportunity analysis occurs on a niche, which is the service responsible for the development of intern software of INESC TEC. The main questions are:

  - How much relevant is to improve performance and flexibility?
  - Can this expand to other REST APIs of INESC TEC?
  - Can this solution be so independent of our business type, that can be used by other organizations?
  - How can we improve the performance and flexibility of the API?

The methods/techniques/tools applied in this key element can be the ones mentioned in the element before. In significant scenarios, it can be used the project charter to set the expectations, resources, and what are the expected outcomes that can provide a guideline in this process.

- Idea generation and enrichment

The improve of REST APIs performance, and flexibility can be made by using a technology/tool capable of giving fast answers to the requests, and that allows each consumer to define the information that needs. However, the use of that technology/tool can be made on different ways: improve the capacity of the REST API, define endpoints for each request with the information need (Bulked REST API) or explore the use of GraphQL.

The mechanisms for communicating core competencies, core capabilities, and shared technologies broadly throughout the corporation and the inclusion of people with different cognitive styles on the idea enrichment team can be used to analyze this key element.

- Idea selection

The Analytic Hierarchy Process (AHP) is a method that allows making complexes decisions based on significant attributes and considering the distinct alternatives. For selecting the idea to apply on this project, it was used this method. It was taking into consideration three different attributes:

  - **Knowledge:** the ability to apply a specific solution;
  - **Efficiency:** how much this solution can improve the problems of the current solution;
  - **Costs:** the implementation of this alternative bring more costs like licenses, specialized hardware, and the necessity of formation.

Thus, it was considering the hierarchy shown in Figure 10.

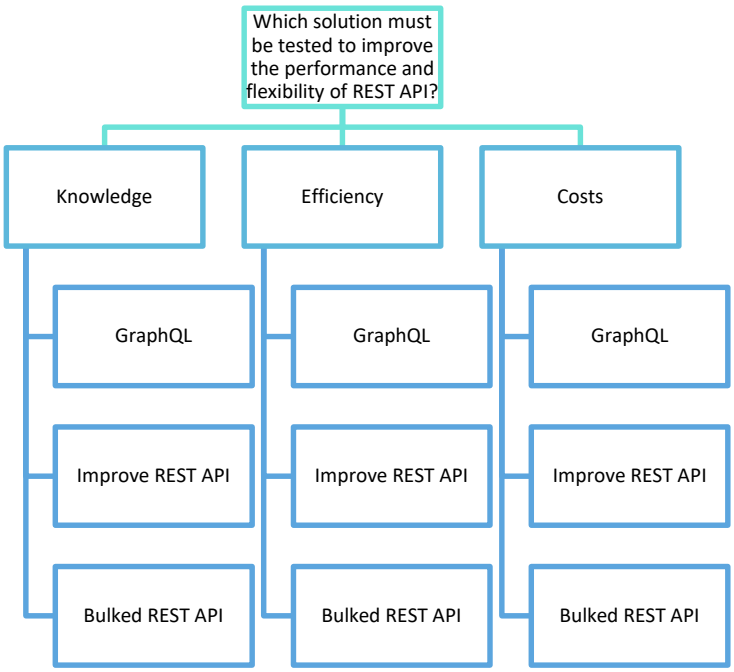


Figure 10 - AHP tree of idea selection

After that, it was defined the priorities among each attribute (Table 2), where it was classified the efficiency more critical, followed by the knowledge and the costs in the end. The scale of comparison used is the one present in Table 1, defined by (Saaty and Vargas, 1991).

Table 1 - Scale for Comparison on AHP, adapted from (Saaty and Vargas, 1991)

Scale	**Degree of preference**
1	Equal importance
3	Moderate importance of one factor over another
5	Strong or essential importance
7	Very strong importance
9	Extreme importance
2,4,6,8	Values for inverse comparison

Table 2 - Comparison matrix of idea selection

	Knowledge	**Efficiency**	**Costs**
Knowledge	1	1/3	5
Efficiency	3	1	7
Costs	1/5	1/7	1

The next step is to normalize the matrix of Table 2 and calculate the relative priority of each attribute (Table 3 and Table 4).

Table 3 - Comparison matrix of idea selection with the sum

	Knowledge	Efficiency	Costs
Knowledge	1	1/3	5
Efficiency	3	1	7
Costs	1/5	1/7	1
SUM	21/5	31/21	13

Table 4 - Comparison matrix of idea selection normalized with a relative priority

	Knowledge	Efficiency	Costs	Relative Priority
Knowledge	5/21	7/31	5/13	0,2828
Efficiency	15/21	21/31	7/13	0,6434
Costs	1/21	3/31	1/13	0,0738

Then, it should evaluate the consistency of the relative priorities (Equation 1).

$$\lambda_{max} = \frac{21}{5} \times (0{,}2828) + \frac{31}{21} \times (0{,}6434) + 13 \times (0{,}0738) = 3{,}0967$$

$$IC = (\lambda_{max} - n) \div (n - 1) = (3{,}0967 - 3) \div (3 - 1) = 0{,}0484$$

$$RC = IC \div 0{,}58 = 0{,}0484 \div 0{,}58 = 0{,}08 < 0{,}1, \textit{so the values are consistent}.$$

Equation 1 - Relative Priorities Evaluation of idea selection

The next phase is the definition of the comparison matrix for each attribute, with each alternative (Table 5, Table 6, and Table 7).

Table 5 - Comparison matrix of idea selection for knowledge

	GraphQL	Improve REST API	Bulked REST API	Priority Vector
GraphQL	1	1/2	1/2	0,5371
Improve REST API	2	1	1	0,2314
Bulked REST API	2	1	1	0,2314

Table 6 - Comparison matrix of idea selection for efficiency

	GraphQL	Improve REST API	Bulked REST API	Priority Vector
GraphQL	1	9	7	0,7791
Improve REST API	1/9	1	5	0,1610
Bulked REST API	1/7	1/5	1	0,0599

Table 7 - Comparison matrix of idea selection for costs

	GraphQL	Improve REST API	Bulked REST API	Priority Vector
GraphQL	1	2	2	0,5
Improve REST API	1/2	1	1	0,25
Bulked REST API	1/2	1	1	0,25

Finally, it is obtained the composed priority for the alternatives and choose the best one (Equation 2).

$$\begin{pmatrix} \mathbf{0,5371} & \mathbf{0,7791} & \mathbf{0,5} \\ \mathbf{0,2314} & \mathbf{0,1610} & \mathbf{0,25} \\ \mathbf{0,2314} & \mathbf{0,0599} & \mathbf{0,25} \end{pmatrix} \times \begin{pmatrix} \mathbf{0,2828} \\ \mathbf{0,6434} \\ \mathbf{0,0738} \end{pmatrix} = \begin{pmatrix} \mathbf{0,6901} \\ \mathbf{0,1875} \\ \mathbf{0,1224} \end{pmatrix}$$

Equation 2 - Calculation of the best alternative of idea selection

The idea selected was to explore GraphQL, which allows to improve performance and flexibility of APIs and study the best integration of this technology on the current architecture.

For analyzing this key element, it can also be considerate the usage of anchored scales, based on technical success probability and strategic fit.

- Concept definition

In the final element, it is going to be found as a solution that incorporates GraphQL, which will improve the performance and flexibility of the REST API.

The methods/techniques/tools that can be used to analyze this key element can be the initial engagement of the customer in real product tests and understanding and determining the performance capability limit of the technology.

2.3.2 Value Offer and Proposition

For any business, the key is the creation of value and its activity based on exchanging a set of products (tangibles) or services (intangibles) that add value for its customers or clients (Nicola et al., 2012).

The value for the customer, for this project, presents the benefits/sacrifices of Table 8.

Table 8 - Benefits/Sacrifices of project

	Product/Service	**Relationship**
Benefits	Product quality Reliability	Technical competence Image Trust
Sacrifices	Price	Time Effort Energy

The solution developed in this project has a price to be used. In exchange, it has quality and, since it is found online, it also offers reliability.

In the relationship domain, the qualified staff, and the image and trust are associated with the use of this solution by an innovation center, which is INESC TEC. However, some time, effort, and energy are needed to apply this solution to the APIs of the organization.

Using a longitudinal perspective (Figure 11), the customer wants a solution to improve its REST APIs, with quality, from an organization that can trust, even though it can be expensive.

At the point of trade, all the benefits are essential since it is needing to have the sacrifice of paying. At post-purchase, the relationship is significant, to aid some difficulties that customer may find during the implementation of the solution, which requires time, effort, and energy. After use, the customer will benefit from all the advantages that the product offers; however, it can be needing to spend some time updating the solution or change some details.

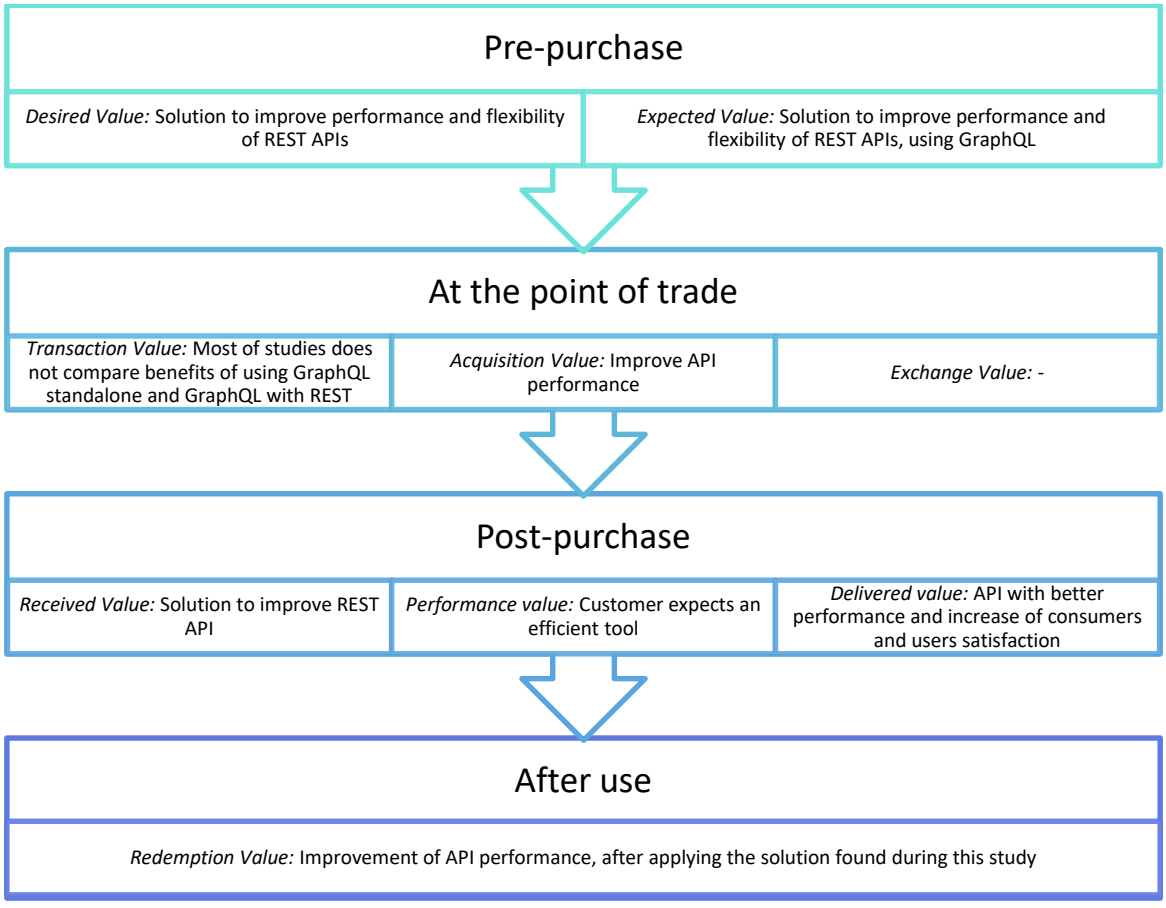


Figure 11 - VC on a longitudinal perspective

Regarding the value proposition, the product developed in this project is a solution for data searches that will use GraphQL, improve the performance and flexibility of the REST APIs, giving to all the consumers an API with the best performance and flexibility and improve the users' satisfaction.

The project presented is an innovative study that can be used by any software development company that wants to improve the performance and flexibility of its APIs, and it uses REST.

2.3.3 Canvas Business Model

When a business idea is built, some questions must be asked, such as:

- Who are the clients?
- What do they value?
- How can we reach them?
- What skills do we need?
- What kind of partnership do we pretend?

The Canvas Business Model is a useful tool to answer those questions systematically (Osterwalder and Pigneur, 2011). This model divided into the following areas:

- **Key partners:** Definition of who are the key patterns/suppliers and what are the motivations for the partnerships;
- **Key activities:** Clarification of what key activities the value proposition requires, and which activities are the most important in distribution channels, customer relationships, and others;
- **Value proposition:** Determination of what value is going to be delivery to the customer and which of his needs is being satisfied;
- **Customer relationship:** Identification of what relationship the target customer expects and how to integrate it on business;
- **Customer segment:** Characterization of the customers;
- **Key resource:** Exposition of what key resources are needed, and which ones are the most important for the different areas;
- **Distribution channel:** Definition of which channels can be used to reach the customers, how much they cost and how they are used;
- **Cost structure:** Exposition of what most cost in the business and which activity/resource is the most expensive;
- **Revenue stream:** Interpretation how much the customers are willing to pay, in which way they want to pay and how much this contributed to the overall revenues.

Figure 12 illustrates the Canvas Business Model created for the project of this Thesis.

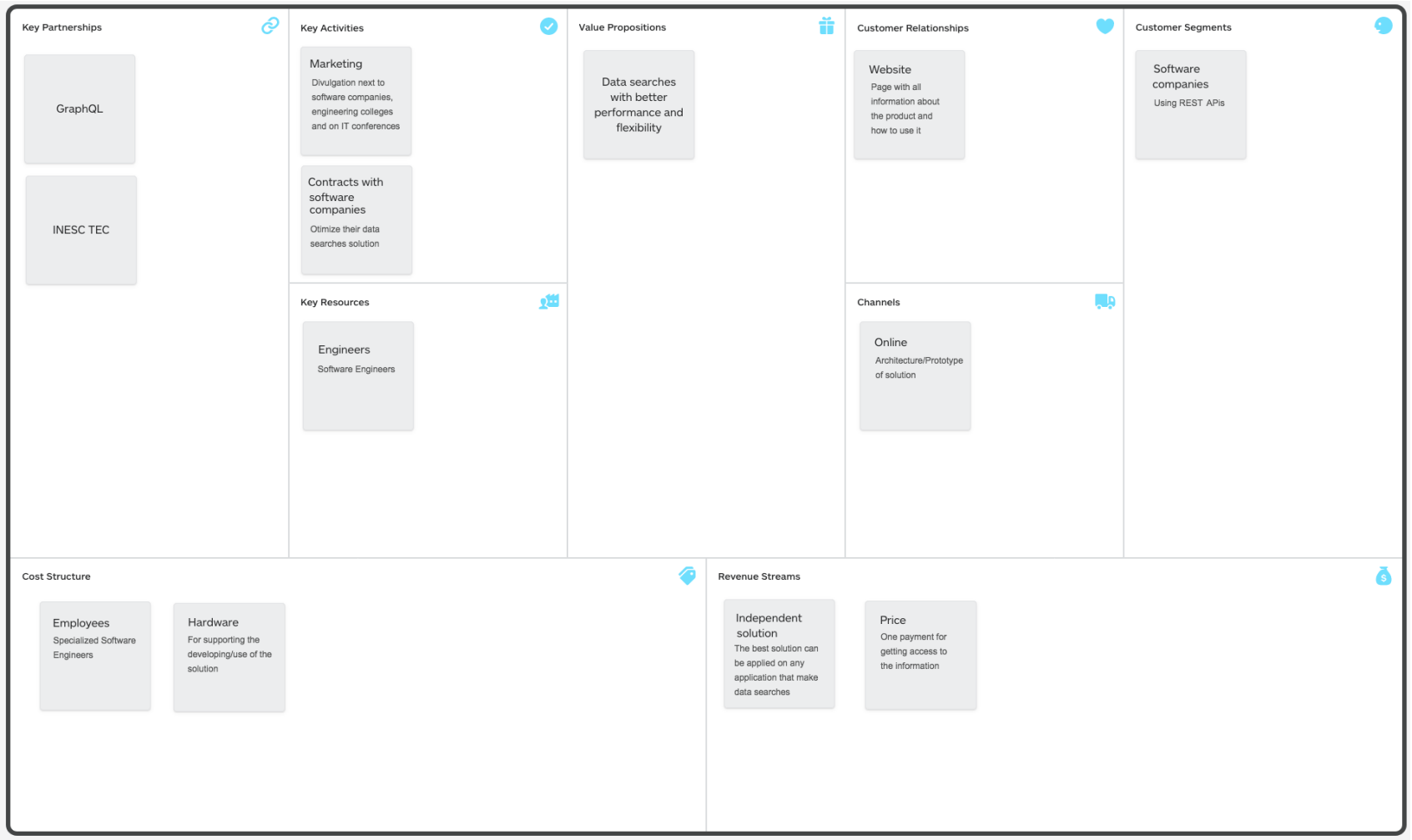


Figure 12 - Canvas Business Model for Analysis of GraphQL Performance: a Case Study

The product developed during this project has one customer segments: the software companies that want to improve their REST APIs.

The customers expected to find this solution on a website, with all the information need and the possibility to clarify some doubts. This product will give them the possibility to improve their APIs and, consequently, improve the consumers and users' satisfaction.

It should be made some marketing in software companies, engineering colleges, and IT conferences to reach the customers.

The INESC TEC is also the key partner since their collaboration is necessary to test the prototypes and define the best solution. GraphQL can be considered a key partner as well because the improvement will be made using their technology.

The main cost structures are the specialized software engineers, and the hardware needs to develop and support the product.

Finally, the solution architecture will be available to the customer by one only payment.

3 State of the Art

Once you stop learning, you start dying.
(Albert Einstein)

This chapter intends to present some essential knowledge about the two technologies that are the subject of study in this thesis: REST and GraphQL. After that, it is described three studies made to compare them and, lastly, the summary of GraphQL adoption by three companies (Netflix, GitHub, and PayPal).

3.1 REST

Representational State Transfer (REST) is an architectural style used in distributed hypermedia systems (Fielding and Taylor, 2000). According to Fielding and Taylor, the constraints of this style are the following ones:

- **Client-Server:** the importance of the separation of concerns principle is represented. The client knows the available services, and it sends requests to the server, which can be executed or declined. That allows applications following this style to have portability and to evolve the components separately;

- **Stateless:** the communication between client and server must have all the necessary information. State date is not stored on the server side. The client is the one who should control all the information in order to be understood by the server. This feature adds visibility, reliability, and scalability to REST;

- **Cache:** it is possible to reuse data from similar previous requests;

- **Uniform Interface:** described as “the central feature that distinguishes the REST architectural style” (Fielding and Taylor, 2000), the definition of the connection established between client and server is made by the identification of resources, its manipulation, self-descriptive messages and hypermedia as the application state system;

- **Layered System:** another constraint to improve scalability as multilayer systems are composed of distinct layers to isolate units by its responsibilities;

- **Code-On-Demand:** the clients can have the ability to download and execute the code on the client side, which is an optional feature of this style.

Nowadays, web services are very used to exchange data among different web systems.

A RESTful web service is a web service implementation that follows the constraints referred to section 3.1. The requests to this type of web services make through Hypertext Transfer Protocol (HTTP) methods, and the information exchange between the client and server can be in eXtensible Markup Language (XML), HTTP or JavaScript Object Notation (JSON).

Usually, the organization is in two distinct forms, present in Figure 13. In the monolithic form, some units are not independent of the core of the application that they belong to, while in the microservice form, there are small independent units with a few sets of responsibilities.

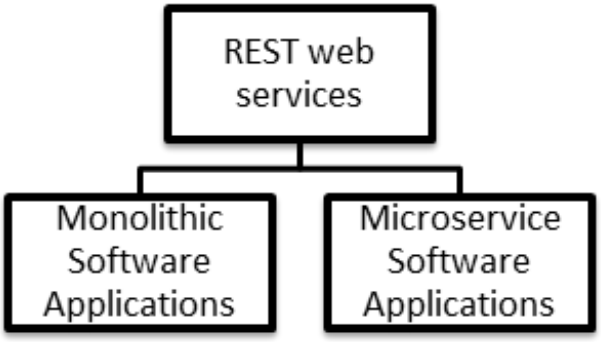


Figure 13 - REST web services types according to (Terzić et al., 2017)

The core principles of a REST API are the following:

- **Resource addressability:** each resource is identified by a Uniform Resource Identifier (URI) and represents a domain concept;

- **Resource representations:** clients work with resource representations;

- **Uniform interface:** for managing resources, it is used the methods defined by the HTTP protocol;

- **Statelessness:** each interaction between client and server are unique;

- **Hypermedia As The Engine Of Application State (HATEOAS):** resources are related to each other, so the client must know the links between them.

3.2 GraphQL

In 2012, Facebook was facing some performance issues on their mobile applications and realized that they needed to optimize how data was sent to client applications, which led to the creation of GraphQL. It was only production-ready, outside the company, in 2016, and it has been used by Facebook, Netflix, GitHub, PayPal, Airbnb, among others (Porcello and Banks, 2018). Figure 14 presents a GraphQL timeline with the main events.

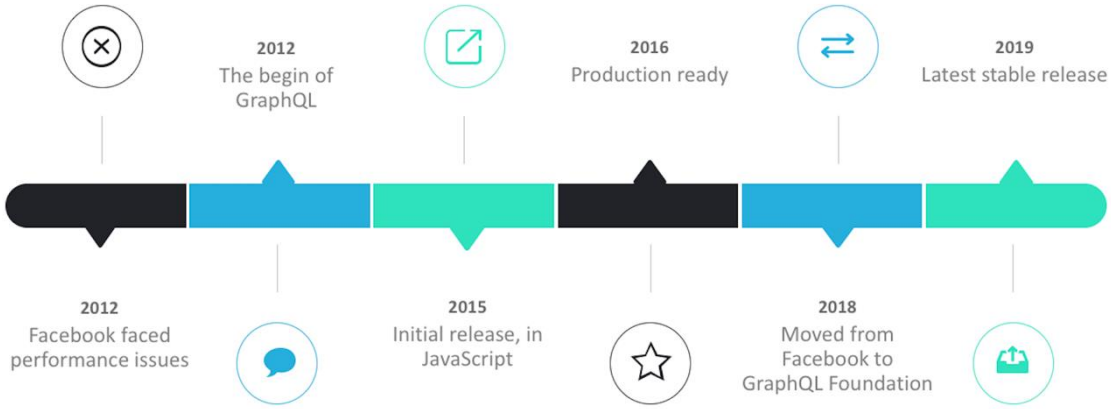


Figure 14 - GraphQL timeline

GraphQL is a query language for APIs, that allows reducing the number of requests need to obtain the desired data, and it grants the components or users to define their data requests and expected responses. This query language, located on the application layer, is transport agnostic, even though is commonly served over HTTP (Porcello and Banks, 2018), and "compatible with any backend that follows the protocol's specification" (Vázquez-Ingelmo et al., 2017).

It also considered as a "specification for client-server communication", and it follows the presented design principles (Porcello and Banks, 2018):

- **Hierarchical:** like a hierarchical graph, the fields used on the queries combine with others;
- **Product-centric:** one of the main goals is to satisfy the data needs of the client;
- **Strong typing:** there is a GraphQL type system, in the GraphQL Server, that validated the schema of the data request;
- **Client-specified queries:** clients can only consume what the GraphQL server allows them to;

- **Introspective:** this language can query the server of GraphQL type system. For example, the GraphiQL online-IDE uses this to allow the developers to get to know the GraphQL schemas (Wittern et al., 2018).

This query language presents a lot of benefits and only a few disadvantages, as it is possible to verify in Figure 15.

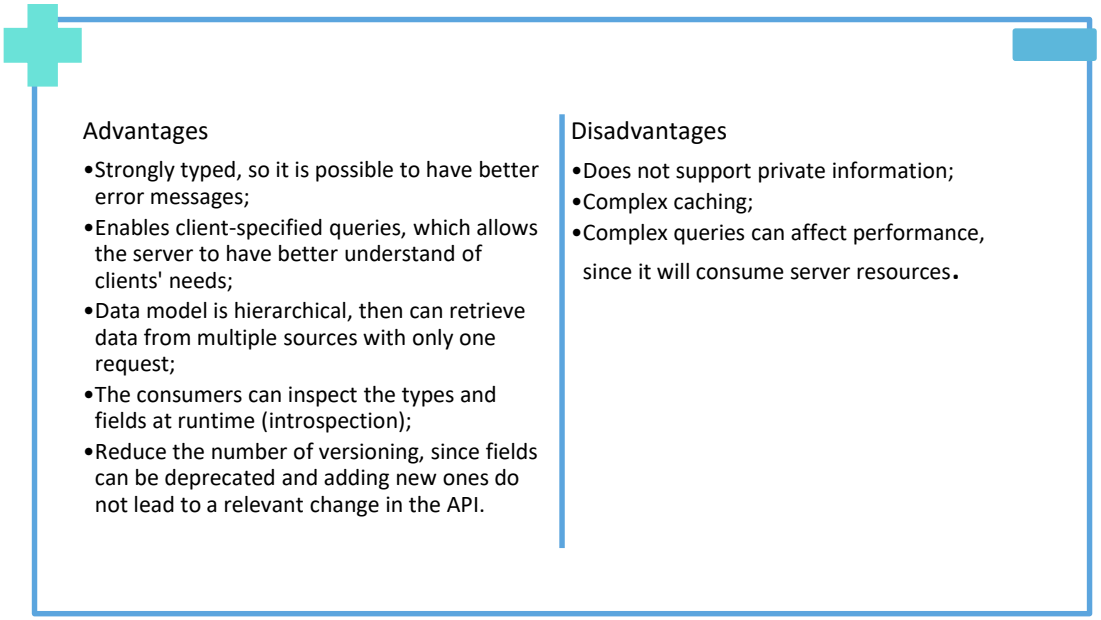


Figure 15 - Advantages and disadvantages of GraphQL, adapted from (Brito et al., 2019)

The fundamental interactions when using GraphQL (Wittern et al., 2018) are visible in Figure 16.

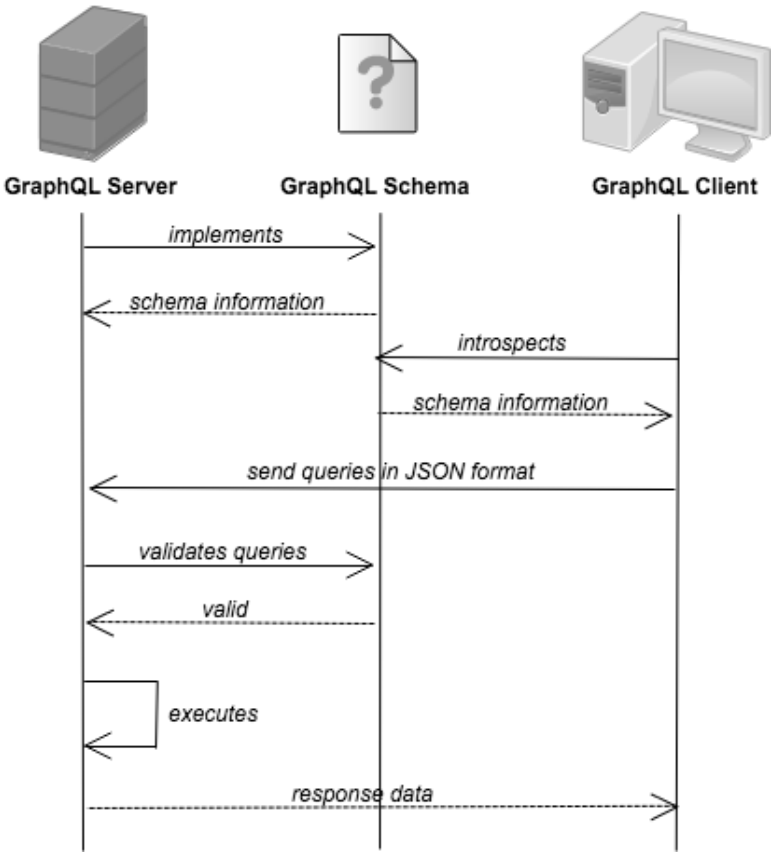


Figure 16 - Interactions of using GraphQL

It is necessary to define a GraphQL Schema, that characterizes the types, relations, and operations allowed to execute with the data. That is how the GraphQL Server knows how and which data will be exposed to the GraphQL Client.

The GraphQL Client can explore the schema in order to acquire knowledge about the exposed data types and the possible operations. Then, using a JavaScript Object Notation (JSON), it will send queries to the server, defining already what operations need to perform and which data is expected to return. In order to obtain data for a specific field, there is a function called *resolver*, that returns the information in the type and shape specified by the schema. They are asynchronous, and it can be used to obtain data from distinct providers: REST APIs, database, or others.

After receiving this request, the GraphQL Server checks if the request agrees with the GraphQL Schema and, in this case, executes it and returns the data to the client or an error.

3.2.1 Schema

The GraphQL Schema is essential because, without it, neither the client or the server knows how to communicate. Thus, this schema design is one of the first steps when implementing a GraphQL API. It is necessary to use the Schema Definition Language (SDL), which is language

and framework agnostic, so it does not matter which technologies are being used to develop the applications. These schemas are text documents that specify the available types and their relations, the entry points, queries, and mutations. Code 1 shows an example of a GraphQL schema.

```
schema {
  query: Query
  mutation: Mutation
}

type User {
  id:     ID!
  userName: String
  name:   String
  projects: [Project]!
}

type Project {
  id: ID!
  name:   String
  responsible: User
  subProjects: [Project]
  team: [User]!
}

type Query {
  listProjects(of: String): [Project]!
  user(userName: String): User
  projectTeam(name: String): [User]!
  users: [User]!
}

type Mutation {
  createProject(responsible: String, name: String): Project
  createUser(userName: String, name: String): User
  addUserToTeam(name: String, userName: String): Project
}
```

Code 1 - Example of a GraphQL schema

At the root of the document, it is described the schema that is divided into:

- **Query:** it is a root GraphQL type because it is the type that maps the operations available to fetch data, which types define on the schema;
- **Mutation:** it is defined the same way as Query, but it has the purpose to write data, following the types described in the schema.

Every GraphQL schema must have a Query type, but it is optional to have a Mutation type. In queries and mutations, the input parameters are named, so the order is not important when a request is made since by specifying the name of the parameter, the system automatically match them.

Queries are used for getting data from a GraphQL API. These describe the data that a client request to a GraphQL server, and it should also specify the units of data by the field which the JSON response has to return. When a query is successful, the return contains the “data” key. In case of unsuccessful, it contains the “error” key and details about the error.

To simplify the process of defining the expected data format and to make it reusable by other queries, it is used fragments. Code 2 shows a request to obtain all the users, expecting only their identifiers (id and userName), which are in the fragment.

```
query {
  users {
          userIdentification
  }
}

fragment userIdentification on User {
  id
  username
}
```

Code 2 - Example of query and fragment

Mutations are defined the same way as queries, but they can cause side-effects since they can change data on the server.

Code 3 reproduces the call of a mutation that will create a project and expected the username of the responsible as return data.

```
mutation {
  createProject(responsible: “someone”, name: “project 1”) {
          responsible {
                  username
          }
  }
}
```

Code 3 - Mutation example

The declared types, the User and Project, are GraphQL Object Types, which means that they are a type with some defined fields. Each type has its fields, like id, userName, and name from User, which indicates that they can only make part of any GraphQL Query that uses the User type. On the example schema, there is also String, which is classified as a scalar type so it cannot have sub-selections on queries. The fields that have an exclamation mark after their types cannot be null. Thus, the GraphQL service has always to return a value when those fields are queried. Finally, the ones that have square brackets represent an array of the type indicated between them; for example, the field projects of User is a list of Project objects.

3.2.2 API creation

GraphQL can be implemented on different programming languages and using distinct frameworks to integrate with. Because of the context of this project, it will be present how a GraphQL API can be developed with Java, more specifically, using Spring Boot (baeldung, 2017) and Maven, since it is the build automation tool used on INESC TEC project.

Firstly, it is necessary to create a Maven Project and include on the dependencies the ones presented in Code 4. With that, Spring Boot will automatically set up the handlers and expose the GraphQL service on the endpoint */graphql*, which accept POST requests, with the GraphQL

Payload on the body. If there is a need to modified the endpoint, that can be done using the *application.properties* file.

```
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-spring-boot-starter</artifactId>
    <version>5.0.2</version>
</dependency>
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java-tools</artifactId>
    <version>5.2.4</version>
</dependency>
```

Code 4 - Maven dependencies required to build a GraphQL API

The library *graphql-java-tools* is responsible for processing the GraphQL Schema files, in order to build the structure and wired the beans that will be used by Spring. The schema files must have the extension *.graphqls,* and they should be in any path inside the project. As mention on 3.2.1, the schema file must have one root query and, optionally, one root mutation across all the schema files, but it can exist more than one file, dividing the schema into modules.

Unlike the schema definition, in the Spring context can be defined more than one bean to handle the root query. Each bean must implement *GraphQLQueryResolver,* and it should exist, on these beans, a method with the same name of every field in the root query of the scheme. Each method must have any parameters that exist on the equivalent of GraphQL schema, and it could also include a final parameter of type *DataFetchingEnvironment*. The example of Code 5 represents the bean that will handle the *listProjects* field on the schema presented in Code 1.

```
public class Query implements GraphQLQueryResolver {
    private ProjectDao projectDao;
    public List<Project> listProjects(string of) {
        return projectDao.listProjects(of);
    }
}
```

Code 5 - Example of bean to handle with the root query field

The system automatically maps all the simple types defined on the GraphQL Schema to their equivalent Java types. The complex types must be represented by a Java bean, which will directly map the fields using the fields name, but the name of the Java class cannot have the name used on the GraphQL type. Code 6 is an example of a complex type defined in Code 1.

```
public class User {
    private String id;
    private String userName;
    private String name;
    private List<Project> projects;
}
```

Code 6 - Example of bean to represent GraphQL complex types

For implementing a mutation, it is applied the logic mention before for the creation of queries, but, in this case, it should be used *GraphQLMutationResolver* instead of *GraphQLQueryResolver*.

Finishing all the implementation and running the Spring Boot application, it is possible to start consuming the GraphQL API.

3.2.3 Adoptions overview

As mentioned in subsection 3.2, some companies are using GraphQL, and some of them have shared the conclusions that achieved since the adoption of this technology. This thesis highlights the experiences of Netflix, GitHub, and PayPal.

Netflix has an internal application, called Monet, responsible for managing the creation and assembly of ads, in order to reach external platforms such as The New York Times and YouTube (Shtatnov and Ranganathan, 2018).

Monet was designed with a React User Interface (UI) layer, accessed by REST APIs. When the evolution of the application started, the use cases become more complex, and it started to appear some problems, like network bandwidth bottlenecks. After some ideas to correct that problems, the use of a middle layer using GraphQL turns out to be the best solution (Figure 17), because of its robust ecosystem and powerful third-party tooling.

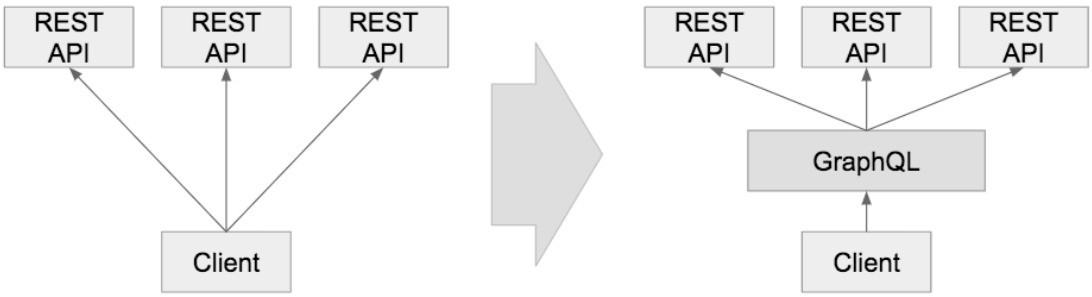


Figure 17 - Architecture of Monet before and after GraphQL, from (Shtatnov and Ranganathan, 2018)

On Netflix blog entry's, it is identified the following benefits, after using that solution for about six months:

- **Main problem solution:** solve network bandwidth bottleneck;
- **Redistributing load and payload optimization:** the server to server calls are very low latency and high bandwidth, improving eight times the performance, comparing with calls made from client to server. With the possibility of client defining the data need for each request, pages started receiving 200KB of data, instead of 10MB;
- **Reusable abstractions:** GraphQL has allowed to defined data and how it relates to the system, so it does not need to worry about business logic related to data join operations;

- **Chaining type systems:** with the definition of entities on GraphQL server, it is easy to generate the TypeScript types, which allows hooking the checks into the build process, preventing issues before deploying wrong code;

- **Developer Interface (DI) / Developer Experience (DX):** using the GraphQL query wrapper, the components implemented on UI only need to describe the data it needs, and the wrapper takes care of all the concerns;

- **Handling failures:** when any resolver of GraphQL query fails, the ones that succeeded return data to the client;

- **Simplify backend end data model:** there is no concern on modeling for the client and, commonly, the backend provides a (Create, Read, Update and Delete) CRUD interface to raw entities;

- **Testing components:** since GraphQL query is translatable into stubs for the test, it is possible to test resolvers from React components.

Even though the reported benefits, there were some problems during the transition, such as the fact that resolvers of GraphQL run isolated, so the network requests were being duplicated. This problem was solved to recur to a caching layer between the resolvers and the REST APIs. Another problem was the fact that GraphQL was not allowing to debug through the browser's network tab.

The solution was to add logs to GraphQL response payload, exposing this information to the client. The last difficulty report was the casting objects, but, since they were using TypeScript, adjusting the methods to require the object properties was smooth.

GitHub has made a total transformation from its REST API to GraphQL, in order to improve its API scalability and API specification (Torikian et al., 2016).

Analyzing their REST API, they concluded that 60% of the requests made to their database tier were made by their API, mainly because of most requests required to navigate through the hypermedia to get all the information need. The second problem was found during the auditing of their endpoints in preparation for an APIv4 since it was tough to collect some meta-information about their endpoints.

The transition began on the backend team of GitHub, especially on the implementation of emoji reactions on comments. After that initial exploration, the frontend team was interested in GraphQL too, and they have achieved a better way to access user data and to improve the efficiency of its presentation on the website.

The main advantages pointed by GitHub are the type of safety, introspection, generated documentation, and predictable responses.

PayPal was another company that introduced GraphQL into its technology stack (Stuart, 2018). The PayPal Checkout had REST APIs, in the beginning, that started to bring problems, since web, mobile apps, and their users were not considered on REST principles'. The increase of round trips from the client to the server, to get all the fundamental data, lead to a processing and rendering time boost as well.

So, the developers built an orchestration API that returns the required data, but this solution also has performance issues. After that, they attempted to build a Bulk REST API, which is a real-time orchestration that allows the clients to delineate the size and shape of the response. However, that solution was not perfect, since it requires that the client have in-depth knowledge about how the APIs work.

Then, they tried GraphQL, and went "all-in" with it, considering that brought productivity to developers, better performance to the app and happiness to the users. The particular aspects considered as the best part of GraphQL adoption are the next ones:

- **Performance:** with one single round trip, it is possible to get precisely the necessary data;
- **Flexibility:** clients, not servers; define the shape and size of data
- **Developer productivity:** the process of learning this technology is easy, and there are useful tools available;
- **Evolution:** it allows them to make better choices when developers are deprecating or evolving their APIs because they are aware of which fields are being used by their clients.

To conclude, these three examples represent how GraphQL as a middle layer or even standalone can bring performance optimization to APIs.

3.3 Comparison between REST and GraphQL

At the first released of GraphQL, some enthusiasts claimed that would be the end of REST, that lead to some studies about how they can be compared and in which cases one can be a better choice than another.

Taking into consideration that GraphQL was built to optimize a REST API, it is easy to point some aspects where this query language exceeds that architectural style.

Three characteristics are evidence as REST drawbacks, according to (Porcello and Banks, 2018), when the comparison is made:

- **Overfetching:** the client does not specify the responses of a REST API, but with the information available that can be useful for distinct consumers, this leads to getting data that is not needed;

- **Underfetching:** even though the response can bring many data, sometimes, the detail of some fields needs the calling of other endpoints and so on, turning the user experience a lot slower;

- **Managing REST Endpoints:** it is common the change of what the client wishes and that leads to an adjustment in the endpoints, between the backend and frontend teams.

Despite this, it is required to go deeper to compare them.

For this thesis, it was analyzed three different studies, that target the comparison between GraphQL and REST, taking into consideration distinct aspects. The choice of these three studies was made based on their relationship with the project context and how they can support the hypotheses evaluated. Stubailo focuses on some properties of an API as resources, Uniform Resource Locator (URL) routes vs. GraphQL schemas and route handlers vs. resolvers (Stubailo, 2017), while Guillen-Drija target some quality attributes like time response, the use of memory, overfetching and others (Guillen-Drija et al., 2018). Vázquez-Ingelmo target the request size and network response times (Vázquez-Ingelmo et al., 2017).

Table 9 presents the similarities and differences found by Stubailo. It was concluded that they share a lot of universal concepts, but some essential aspects may dictate the use of one instead of the other.

Table 9 - Similarities and differences between REST and GraphQL found by Stubailo

		REST	**GraphQL**
Resources	*Identification*	Yes	Yes
	HTTP Usage	Yes	Yes
	JSON Response	Yes	Yes
	Object Identify	Endpoint	Separate from how is fetch
	Determination of Shape and Size	Server	Client
URL routes vs. GraphQL schemas	*List of Operations*	List of Endpoints	List of fields (at Query and Mutation)
	The distinction between Reading and Writing	Yes	Yes
	Multiple Calls to Relate Resources	Yes	No
	First-class Concept	No	Yes
	Modify Reading into Writing/ Writing into Reading	HTTP verbs	Keyword in the query
Route handlers vs. resolvers	*Function Call*	Endpoints	Fields
	Handle Networking Boilerplate	Using frameworks or libraries	Using frameworks or libraries

		REST	GraphQL
	Number of Handler/ Resolver Calls	One by each request	Many by each query
	Response Build	By developer	By GraphQL execution library

GraphQL looks good to implement an API quickly, that can return complex data, decreasing the time waste in the implementation of multiple endpoints to be able to respond to the client requests. However, on the other hand, REST is already mature, with many tools and integrations. The results of the study made by Guillen-Drija are in Table 10.

Table 10 - Similarities and differences between REST and GraphQL found by Guillen-Drija

Sub characteristic	**Metric**	**REST**			**GraphQL**		
		Average	***Standard deviation***	***Error***	***Average***	***Standard deviation***	***Error***
Temporal behaviour	***Response time***	11,13	3,77	0,69	16,23	4,22	0,77
	Throughput (calls)	149,63	11,05	2,02	190,7	7,01	1,28
Resources usage	***Memory usage (bytes)***	588,6	65,69	11,99	156,33	5,37	0,98
	Cache usage						
	Overfetching (bytes)	688,67	116,28	21,23	79,33	15,49	2,83
	Underfetching (made calls)	4	0,68	0,12	1	0	0
Capacity of software	***Capacity (answered calls/made calls)***	1			1		
	Speed under stress (ms)	970,17	123,36	22,52	1138,97	88,74	16,20

The main conclusions are:

- Because of the cache usage in REST, usually, this style will present a better speed on response time than the GraphQL;
- With the detail that the requests made with GraphQL can have, this language presents an advantage on underfetching and overfetching;
- On atomic calls, REST offers better speed of response;
- The GraphQL handles better with the use of memory.

Considering the network response time represented on Figure 18 and the total size of 13,90 KB (GraphQL) against the 26,66 KB (REST), the implementation of the GraphQL API improves the performance of the system and reduces the memory usage.

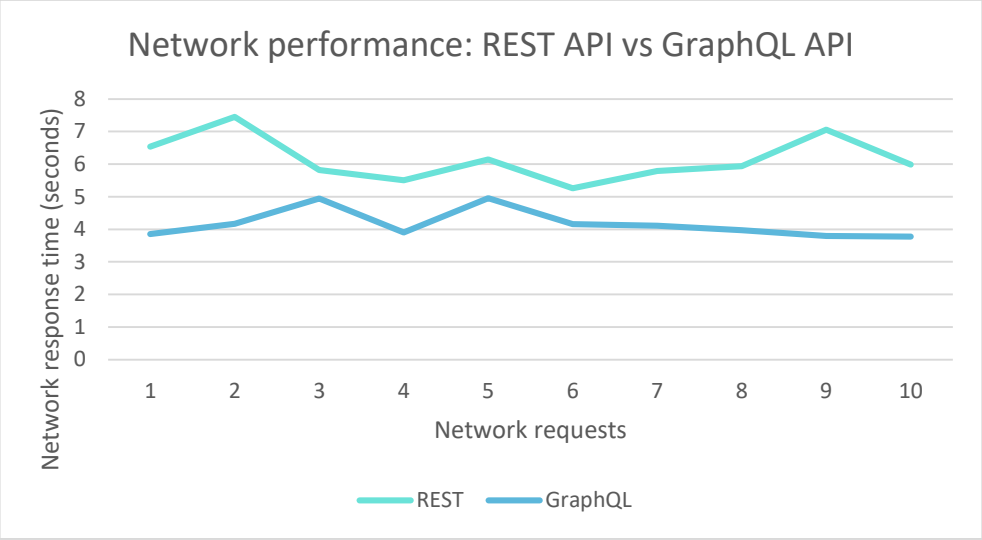


Figure 18 – Graphic with network performance results obtain by Vázquez-Ingelmo

In conclusion, even though the two technologies share some similar points and it cannot be stated as replacement of each other, GraphQL presents better flexibility, response times, and use of memory comparing with REST.

3.4 Adopted technologies

This subsection exposes some technologies involved in the implementation of the solution to this project. The aim is to give the reader a brief presentation about them, not to concentrate on the study of the available technologies that could use in this context. All the choices are based on the developer's experience and the technological stack of the institution.

OpenAPI

The OpenAPI specification, which is the new name for the Swagger specification, is a specification for machine-readable that allows to produce and consume REST APIs (Open API Initiative, 2017).

Its files are commonly used by other applications, for example, the Swagger-Codegen, to generate code and some code documentation automatically. The use of this specification can help on the design of applications and improve the development time and costs since it is possible to generate the code.

The snippet Code 7 presents an example of an OpenAPI specification file. This file can be on JSON, or YAML Ain't Markup Language (YAML) format and presents a hierarchy form.

```
{
  “swagger”: “2.0”.
  “info”: { (…) },
  “host”: “petstore.swagger.io”,
  “basePath”: “/api”,
  “schemes”: [
          “http”
  ],
  “consumes”:     [
          “application/json”
  ],
  “produces”: [
          “application/json”
  ],
  “paths”: {
          “/pets”: {
                  “get”: {
                          “description”: “Returns all pets from the system that the user
has access to”,
                          “produces”: [
                                  “application/json”
                          ],
                          “responses”: {
                                  “200”: {
                                          “description”: “A list of pets.”,
                                          “schema”: {
                                                  “type”: “array”,
                                                  “items”: { (…) }
(…)
```

Code 7 - Example of OpenAPI Specification, adapted from (Open API Initiative, 2017)

Spring

Spring is an open source framework for Java, created by Rod Johnson, based on some patterns dependency injection and inversion of control (Pivotal, 2019). Several modules compose it, so this framework is used for a diversity of services like aspect-oriented programming, authentication and authorization, data access, messaging, testing, remote access, and others.

On version 5.0, it adds end to end support for reactive and servlet-based apps on the Java Virtual Machine (JVM) and provides some streamlining on developing modern applications. It also allows to integrate with GraphQL and improve the implementation of APIs using it, using Spring Boot, which makes easy to create stand-alone Spring-based applications ready to execute with fewer configurations.

GraphQL Java

GraphQL-Java is a library to implement a GraphQL Server using Java, and it bases on GraphQL Java Engine, which focuses on executing queries (Marek and Baker, 2019). This library goes further than that, and with the GraphQL Java Spring Boot adapter, it is possible to expose the GraphQL API via Spring Boot, over HTTP. It requires at least the Java version 8, and it is available on Maven Central repository.

Swagger2GraphQL

Swagger2GraphQL is a tool that helps to convert an existing Swagger schema (OpenAPI specification) to GraphQL types, setting the resolvers with HTTP calls to the endpoints of the

REST API. By consulting the GraphQL API, it is possible to extract the GraphQL schema generated from the conversion (Krivtsov, 2016).

4 Design

Every adventure requires a first step.
(Cheshire Cat)

In this chapter is presented the prototypes' architecture and possible alternatives, taking in consideration the requirements defined.

All the designs shown in this section has the goal to achieve a technologic solution for the idea present in Figure 19.

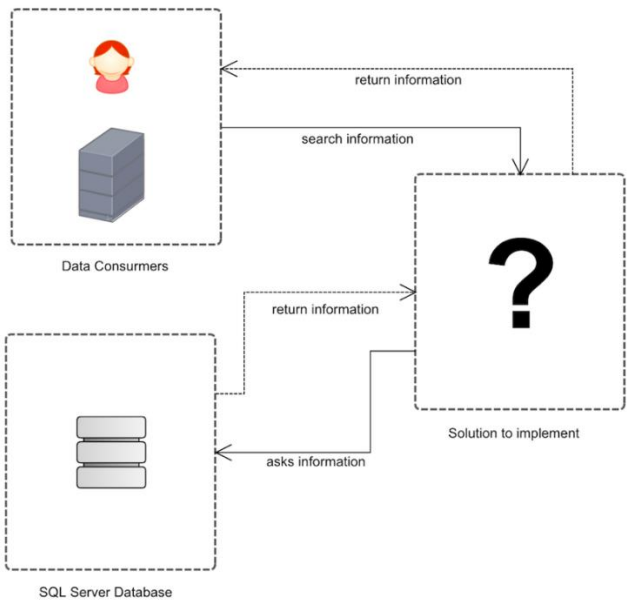


Figure 19 - Solution idea

4.1 Requirements Analysis

Requirements are a specification of what the system must contain and how it should behave, including the user and developer's views (Wiegers and Beatty, 2013).

The requirements analyze as the main goal to understand and to document what the stakeholders need, so the product that will be delivery matches the expectations.

Without this process, it is challenging to design software that will correspond to what the client asks for.

At this chapter, it is defined the system actors and the requirements of this project.

4.1.1 System Actors

The identification of system actors is essential to define their necessities. However, the application user is commonly defined as there is a "monolithic group with similar characteristics and needs" (Wiegers and Beatty, 2013). It must be defined the multiple user classes, their roles, and privileges.

The present thesis wants to analyse the performance of data searches, and it will apply its tests to an application that already exists: IRIS. When we are talking about the system actors of an API, it is possible to classify them into two classes:

- **Direct data consumers:** the ones that see information through the API result (JSON format, for example);
- **Indirect data consumers:** users that access the data through other applications that integrated with this one.

On IRIS case, these users can present distinct privileges because of its roles. On projects module, a controller and a project responsible will access all the information of the project, but a team member will only get necessary information about it so that the complexities of the request can be affected by users' roles. Even though these differences, for this thesis, they share the same goal: obtain information.

4.1.2 Requirements

The functional requirements describe the main functionalities that the system must offer to the identified system actors.

Taking that into consideration and due to the primary goal of this thesis, the solution developed only have one main requirement: allow data searches. Table 11 identifies the requirement of the solution and possible subdivisions.

Table 11 - Functional requirements

Identification	Description
FR01	The user shall be able to search for data about projects, by their attributes in a fast way.
FR01.1	The user shall be able to search direct data about projects (data available on the project entity), by their attributes in a fast way.
FR01.2	The user shall be able to search indirect data about projects (data available on other entities related to the project entity), by their attributes in a fast way.

Therefore, Figure 20 represents the only functional requirement of the solution.

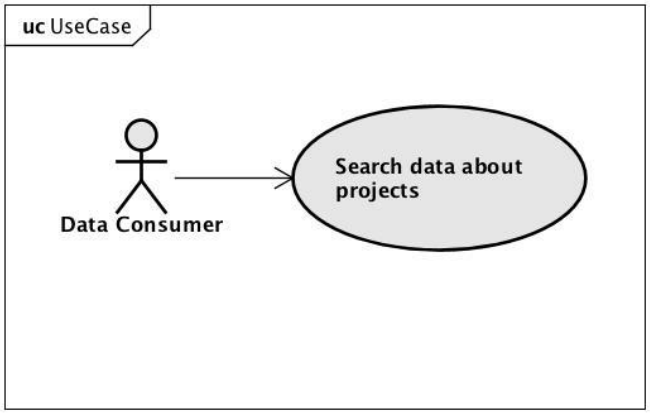


Figure 20 - Solution use case

In addition to the functional requirements, it can also exist other requirements that should be taking in consideration during the development of the system, in order to guarantee some particularities related with data, external interfaces or quality attributes, for example.

As a data requirement, it is necessary that the solution consumes the information presented in a Structured Query Language (SQL) Server database.

As external interface requirement, data is requested via HTTP requests.

As quality attributes, it must consider the following ones:

- **Performance:** the developed solution shall answer to each request with the best time possible. It excepted a short response time when it is a simple request;

- **Safety:** since it will be some sensitive information involved, the solution shall only provide information after authentication;

- **Interoperability:** the solution shall be able to be consulted by the IRIS UI.

4.2 Possible Approaches

To study the performance and flexibility of GraphQL APIs comparing with REST APIs, there are a diversity of possible approaches. Taking into consideration the previous knowledge stated in the chapter 3, it can be idealized three distinct approaches (Figure 21):

- **GraphQL standalone:** assuming the ideas that GraphQL represents globally a better performance and flexibility than REST, a GraphQL API would be able to improve these measures;

- **REST with GraphQL:** analyzing the GraphQL adoption by Netflix, the integration of the two technologies can improve the current solution;

- **GraphQL on complex queries + REST on simple queries:** the studies of Guillen-Drija and Vázquez-Ingelmo support this approach, since simple requests have better performance using REST and complex queries, where multiple endpoints are needed, presents better performance using GraphQL. However, this approach can lead to one of the advantages of GraphQL, shown in subsection GraphQL, that complex queries can consume many server resources.

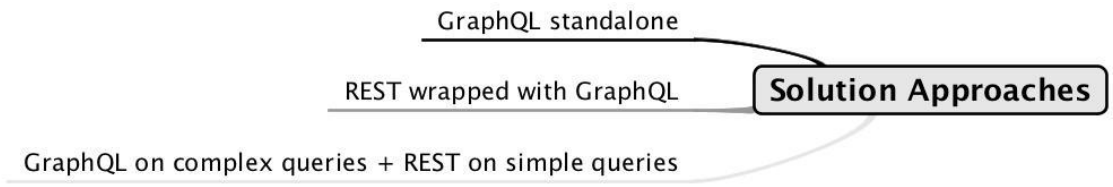


Figure 21 - Solution approaches mind map

For selecting the solution approaches to apply on this project, it was used the AHP method[1]. It was taking into consideration three different attributes:

- **Architectural Complexity:** the complexity that the approach presents;

[1] The AHP is a method that allows making complexes decisions, based on significant attributes and considering the distinct alternatives.

- **Efficiency:** how much this approach can contribute to finding a better solution;
- **Time:** the time needs to implement the approach since the time for developing the thesis is limited.

Thus, it was considering the hierarchy shown in Figure 22.

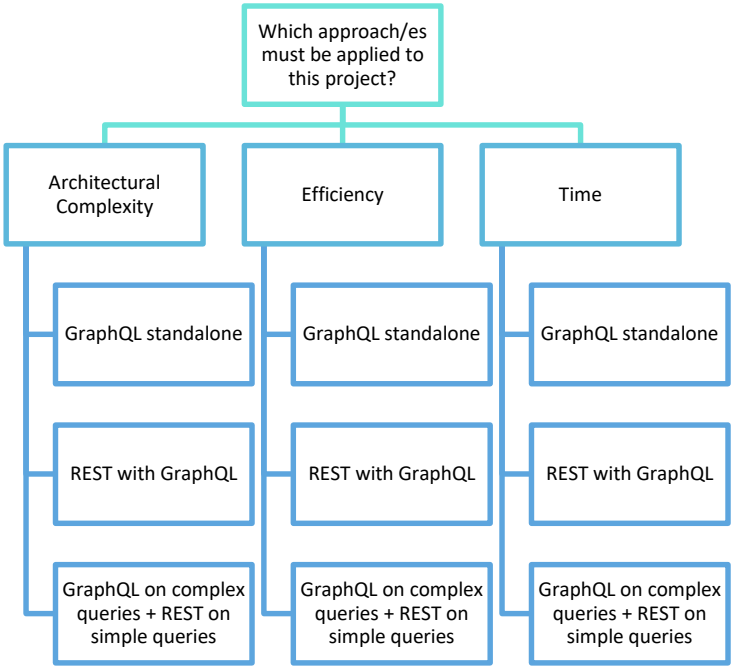


Figure 22 - AHP tree of approaches selection

The definition of the priorities among each attribute is presented in Table 12, where it was classified the efficiency more critical, followed by the architectural complexity and the time in the end. The scale of comparison used is the one present in Table 1, on subsection Business and Innovation Process.

Table 12 - Comparison matrix of approaches selection

	Architectural Complexity	**Efficiency**	**Time**
Architectural Complexity	1	1/3	5
Efficiency	3	1	7
Time	1/5	1/7	1

The next step is to normalize the matrix of Table 12 and calculate the relative priority of each attribute (Table 13 and Table 14).

Table 13 - Comparison matrix of approaches selection with the sum

	Architectural Complexity	Efficiency	Time
Architectural Complexity	1	1/3	5
Efficiency	3	1	7
Time	1/5	1/7	1
SUM	21/5	31/21	13

Table 14 - Comparison matrix of approaches selection normalized with a relative priority

	Architectural Complexity	Efficiency	Time	Relative Priority
Architectural Complexity	5/21	7/31	5/13	0,2828
Efficiency	15/21	21/31	7/13	0,6434
Time	1/21	3/31	1/13	0,0738

Then, it should evaluate the consistency of the relative priorities (Equation 3).

$$\lambda_{max} = \frac{21}{5} \times (0{,}2828) + \frac{31}{21} \times (0{,}6434) + 13 \times (0{,}0738) = 3{,}0967$$

$$IC = (\lambda_{max} - n) \div (n - 1) = (3{,}0967 - 3) \div (3 - 1) = 0{,}0484$$

$$RC = IC \div 0{,}58 = 0{,}0484 \div 0{,}58 = 0{,}08 < 0{,}1, so\ the\ values\ are\ consistent.$$

Equation 3 - Relative Priorities Evaluation of approaches selection

The next phase is the definition of the comparison matrix for each attribute, with each alternative (Table 15, Table 16, and Table 17).

Table 15 - Comparison matrix of approaches selection for architectural complexity[2]

	GraphQL standalone	REST with GraphQL	GraphQL on complex queries + REST on simple queries	Priority Vector
GraphQL standalone	1	4	9	0,7132
REST with GraphQL	1/4	1	4	0,2199
GraphQL on complex queries + REST on simple queries	1/9	1/4	1	0,0669

Table 16 - Comparison matrix of approaches selection for efficiency

	GraphQL standalone	REST with GraphQL	GraphQL on complex queries + REST on simple queries	Priority Vector
GraphQL standalone	1	5	9	0,7352
REST with GraphQL	1/5	1	4	0,1994
GraphQL on complex queries + REST on simple queries	1/9	1/4	1	0,0654

[2] In the matrix for architectural complexity, the bigger the number, the simpler is the architecture of the approach.

Table 17 - Comparison matrix of approaches selection for time

	GraphQL standalone	REST with GraphQL	GraphQL on complex queries + REST on simple queries	Priority Vector
GraphQL standalone	1	1	5	0,4545
REST with GraphQL	1	1	5	0,4545
GraphQL on complex queries + REST on simple queries	1/5	1/5	1	0,2121

Finally, it is obtained the composed priority for the alternatives and choose the bests ones (Equation 4).

$$\begin{pmatrix} \mathbf{0,7132} & \mathbf{0,7352} & \mathbf{0,4545} \\ \mathbf{0,2199} & \mathbf{0,1994} & \mathbf{0,4545} \\ \mathbf{0,0669} & \mathbf{0,0654} & \mathbf{0,2121} \end{pmatrix} \times \begin{pmatrix} \mathbf{0,2828} \\ \mathbf{0,6434} \\ \mathbf{0,0738} \end{pmatrix} = \begin{pmatrix} \mathbf{0,7083} \\ \mathbf{0,2240} \\ \mathbf{0,0767} \end{pmatrix}$$

Equation 4 - Calculation of the best alternative of approaches selection

The approaches selected were the GraphQL standalone and the REST with GraphQL.

4.3 Prototyping

Prototyping is the phase that follows the requirements analysis and solution design (Rettig, 1994). In this context, the prototype is the implementation of simple functionalities to evaluate the advantages and disadvantages of the implemented architecture/technology, with a focus on performance.

Since this thesis aims to study the performance of solutions with GraphQL and compare them with the one that uses REST (current solution), it was decided to make two different prototypes with distinct and alternative architectures. Because of the previous knowledge and the technologies' stack of the team where these prototypes are implemented, Java and the Spring framework will be used for the development of GraphQL API.

All the information used by the prototypes will be available on a SQL Server database, which is the one used on INESC TEC and consume some data of the tables present on Figure 23, which is a short and incomplete view of the database model of the Projects module.

The *Project* entity is the main one where all the information that classified the project is contained. The *Typology* entity is an institution classification that allows organizing some information about the programs, financing and founds involved on the project. The *FinancingEntity* entity is one of the aspects that constitutes the *Typology* and represents the entity that is financing the project. The *TypologyClass* entity is also part of the *Typology* and represents a generic aggrupation of the typologies, accordingly with the sources. Finally, the *Team* entity is the definition of the responsible and the internal order where the costs will be imputed, inside each INESC TEC research center that works in the project.

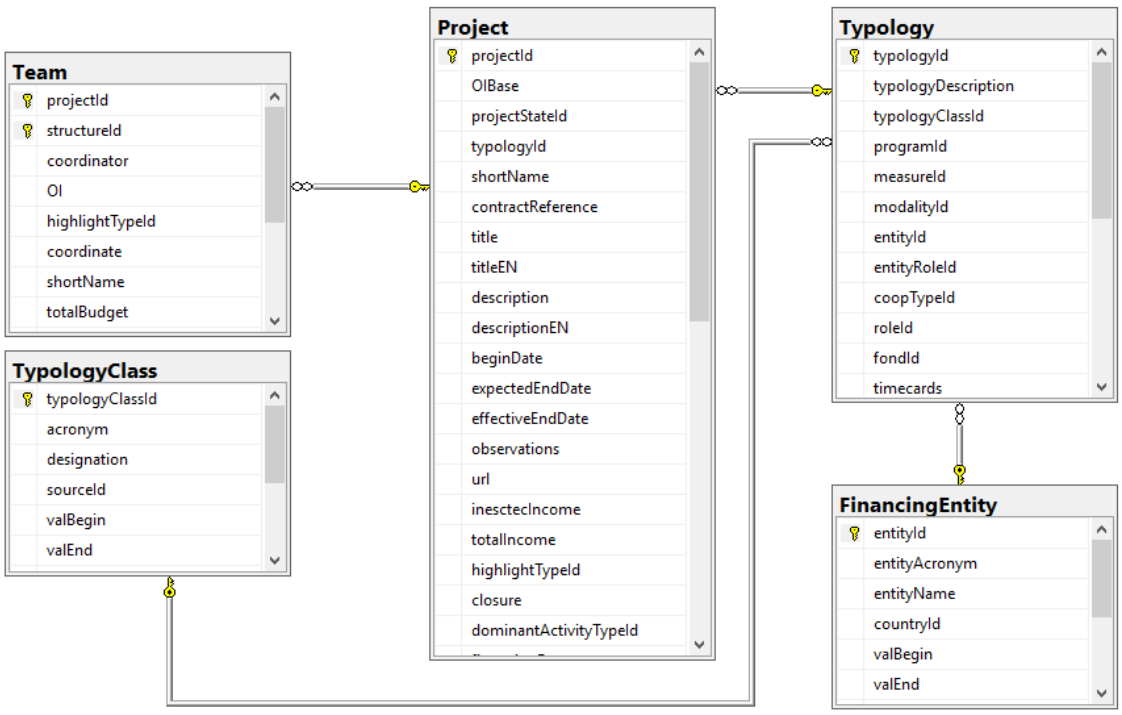


Figure 23 - Database model for the projects module

4.3.1 Prototype 1 – GraphQL standalone

The prototype defined as 1 is the solution that aims to replace the use of REST API for a GraphQL API when data searches are done. Since the current solution follows the OpenAPI specification, this file can be used to convert the existing API into a GraphQL schema, using the tool Swagger2GraphQL.

This solution has three components (Figure 24):

- **Projects (GraphQL-Spring API):** which is the API responsible for receiving the requests by data consumers and deal with them and to authenticate the clients;
- **Security (Spring Security):** which is the module responsible for the authentication and access control of the API;
- **SQL Server Database (Database):** which is the database that contains the information available. This component already exists; it will be used for data querying by the prototype.

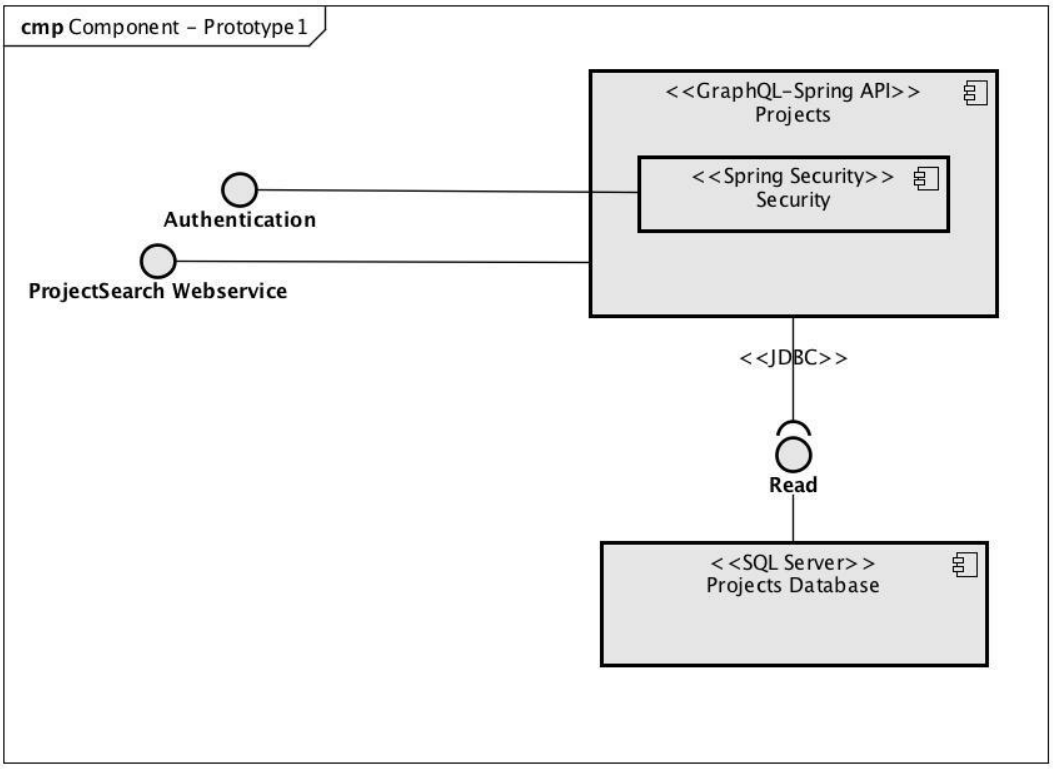


Figure 24 - Component diagram of Prototype 1

These three components will be on distinct servers (Figure 25):

- **GraphQL Spring Server (Tomcat Server):** is the solution server and it will be using Tomcat;
- **Security (Spring Security):** is the module responsible for the authentication and access control, running on Tomcat too;
- **Database Server (Microsoft Server):** the server where the database is.

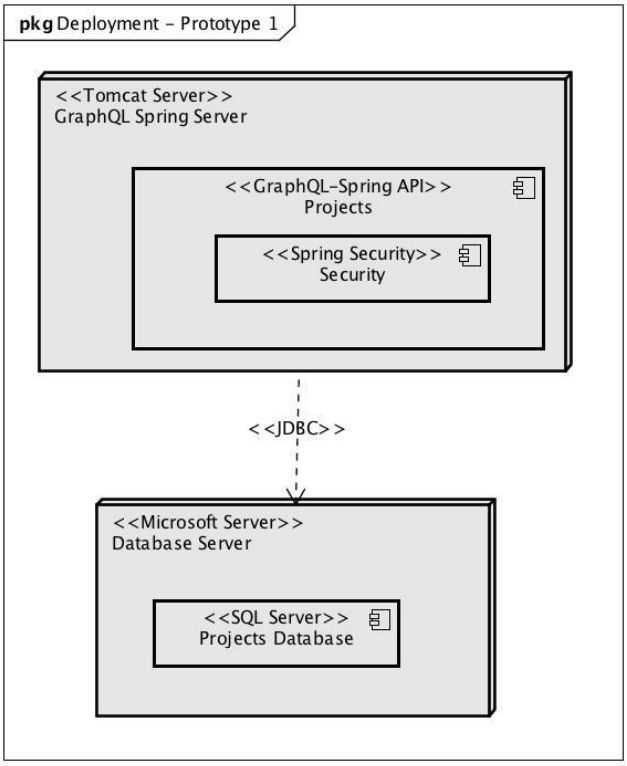


Figure 25 - Deployment diagram of Prototype 1

The Projects API will have distinct layers (Figure 26), in order to define responsibilities and organized the implementation:

- **Service:** responsible for receiving an answer to the HTTP requests and is composed of two segments:
    - **Resolver:** the one that defines which query will execute and what data is necessary, that is the reason why it communicates with the Repository segment;
    - **Exception Handler:** controls the exceptions that may occur and treats them, so can be used by Resolver to answer to some requests.
- **Data:** in control of all the data management and presents the next segments:
    - **Entity:** has the types defined on GraphQL schema and mapped to Java's objects;
    - **Repository:** deals with the operations with the database, to obtain all the necessary data, on the formats defined on Entity.
- **Security:** responsible for authentication and access control management.

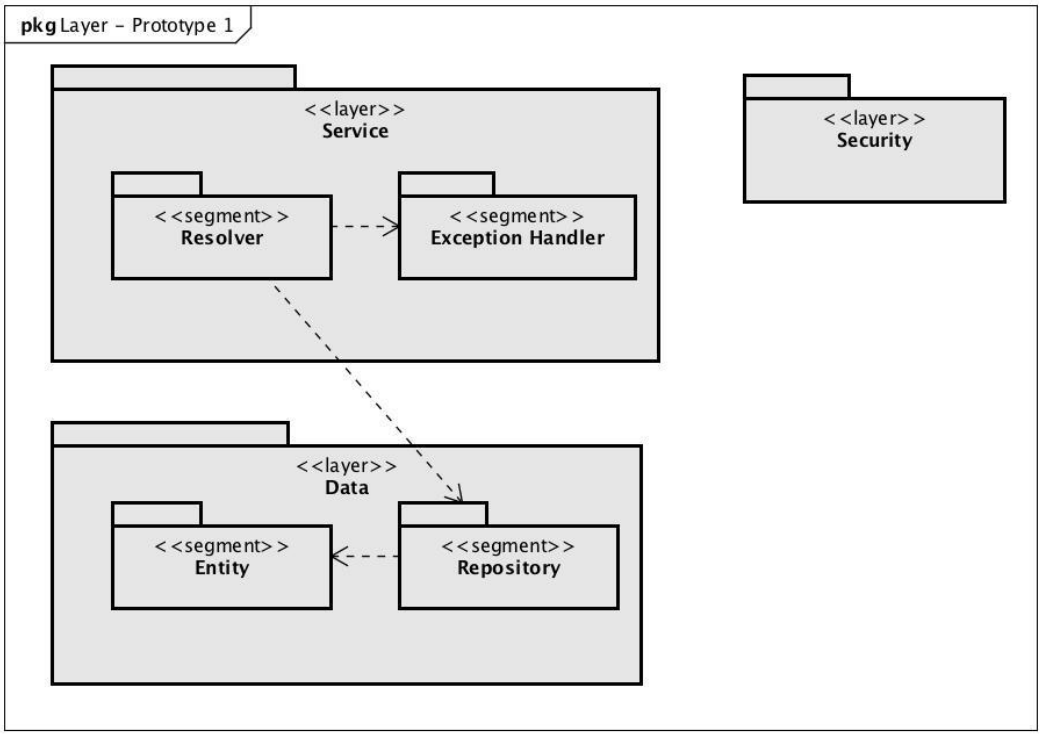

pkg Layer – Prototype 1
<<layer>>
Service
<>
Resolver
<>
Exception Handler
<<layer>>
Security
<<layer>>
Data
<>
Entity
<>
Repository

Figure 26 - Module layer of Prototype 1

The access to the database it will be made using Spring Data so it can be used the CRUD repositories that automatically generate the methods find, save, count, and delete for each entity. Thus, the implementation of resolvers will be simple; only have to call the repositories of each entity. The module Spring Data was considered since it is a layer that already exists and can be reused.

In conclusion, for developing the GraphQL API, it will be used some tools to optimize the implementation, such as Spring and Swagger2GraphQL.

4.3.2 Prototype 2 – REST with GraphQL

The prototype defined as 2 is the solution whose goal is to use a middle layer, between the REST API and the consumer, with a GraphQL API, when data searches are done. Like prototype 1, Swagger2GraphQL will be used to create the GraphQL schema, but it will not be made any adjustments, because this API will work as an intermediate between the REST API and the client.

The solution will be composed of three components (Figure 27):

- **Projects_GraphQL (GraphQL-Spring API):** which is the API responsible for receiving the requests by data consumers and ask for data to the Projects_REST. It will also authenticate the clients;

- **Projects_REST (REST-Spring API):** which is the current solution and it will answer to a request made by Projects_GraphQL;

- **Security (Spring Security):** which is the module responsible for the authentication and access control of each API;

- **SQL Server Database (Database):** which is the database that contains the information available.

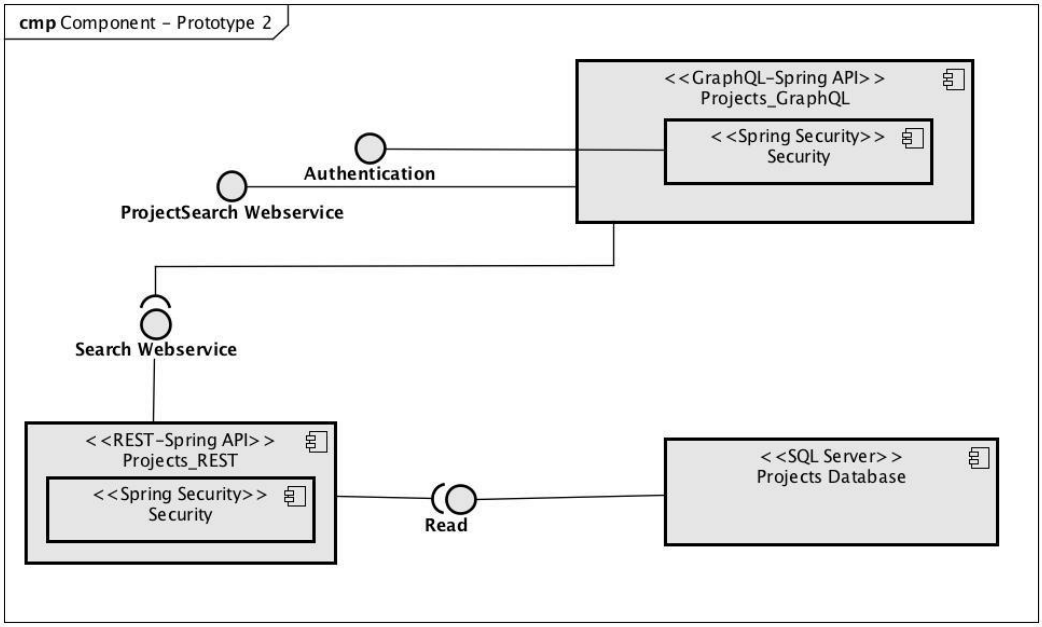


Figure 27 - Component diagram of Prototype 2

These three components will be on distinct servers (Figure 28):

- **GraphQL Spring Server (Tomcat Server):** is the solution server and it will be using Tomcat;

- **REST Spring Server (Tomcat Server):** is the current solution server and it will be using Tomcat;

- **Security (Spring Security):** is the module responsible for the authentication and access control, running on Tomcat too;

- **Database Server (Microsoft Server):** the server where the database is.

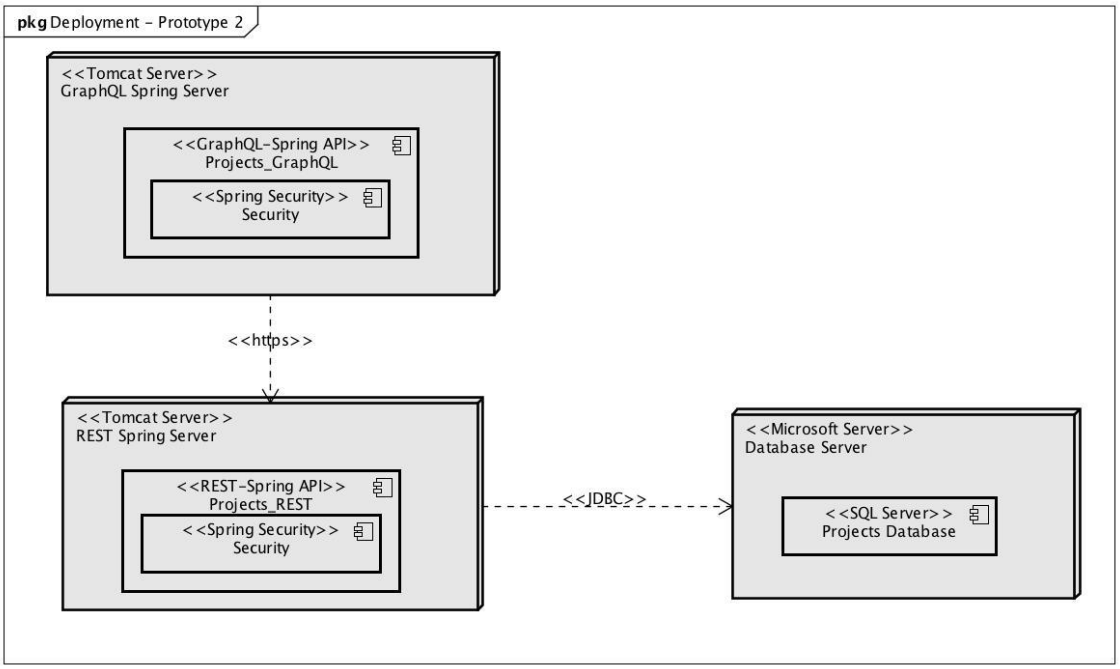


Figure 28 - Deployment diagram of Prototype 2

The **Projects_GraphQL** API will have two layers (Figure 29) since it is only an intermediate:

- **Service:** responsible for receiving an answer to the HTTP requests and is composed of two segments:
    - **Resolver:** the one that defines which query will execute and which endpoint of Projects_REST should be called;
    - **Exception Handler:** controls the exceptions that may occur and treats them, so can be used by Resolver to answer to some requests.
- **Security:** responsible for authentication and access control management.

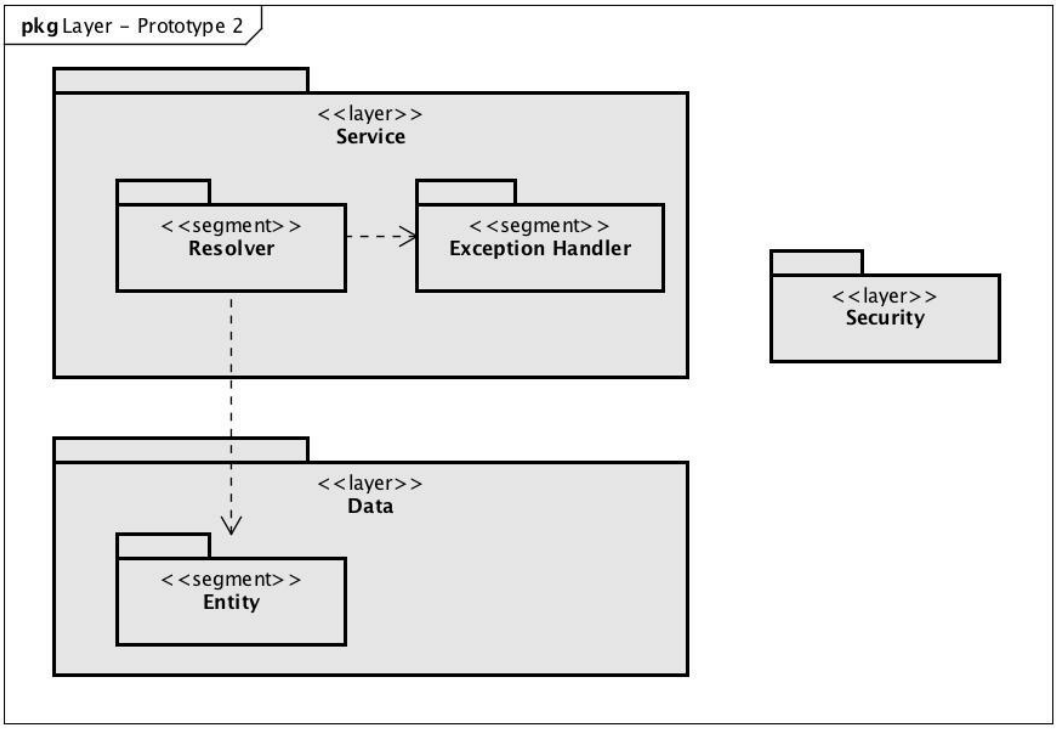

pkg Layer – Prototype 2
<<layer>>
Service
<>
Resolver
<>
Exception Handler
<<layer>>
Security
<<layer>>
Data
<>
Entity

Figure 29 - Module Layer of Prototype 2

To conclude, for developing the GraphQL wrapper, it will be used the same tools of prototype 1, but there are some differences in architecture and the responsibility of GraphQL API.

5 Implementation

It always seems impossible until it is done.
(Nelson Mandela)

At this chapter, it explains some details about the implementation of the prototypes specified in the chapter before. For developing the prototypes were taking into consideration some aspects in order to produce a scalar, flexible, and secure prototypes, following the good practices of software development.

5.1 Common implementation

Even though it is developed two distinct prototypes with different architectures; there are some similar tasks during their development.

One of them is the generation of a GraphQL Schema that represents the model used on the REST API and the queries available. As mentioned in the section before, the current REST API follows the OpenAPI specification so that it can be consulted about all the operations available in the API (Figure 30). This specification can be represented as well as a JSON file (Code 8) and through there can identify the endpoints of the application, which are represented by the path tag, and also the entities involved, that are defined inside the definitions tag.

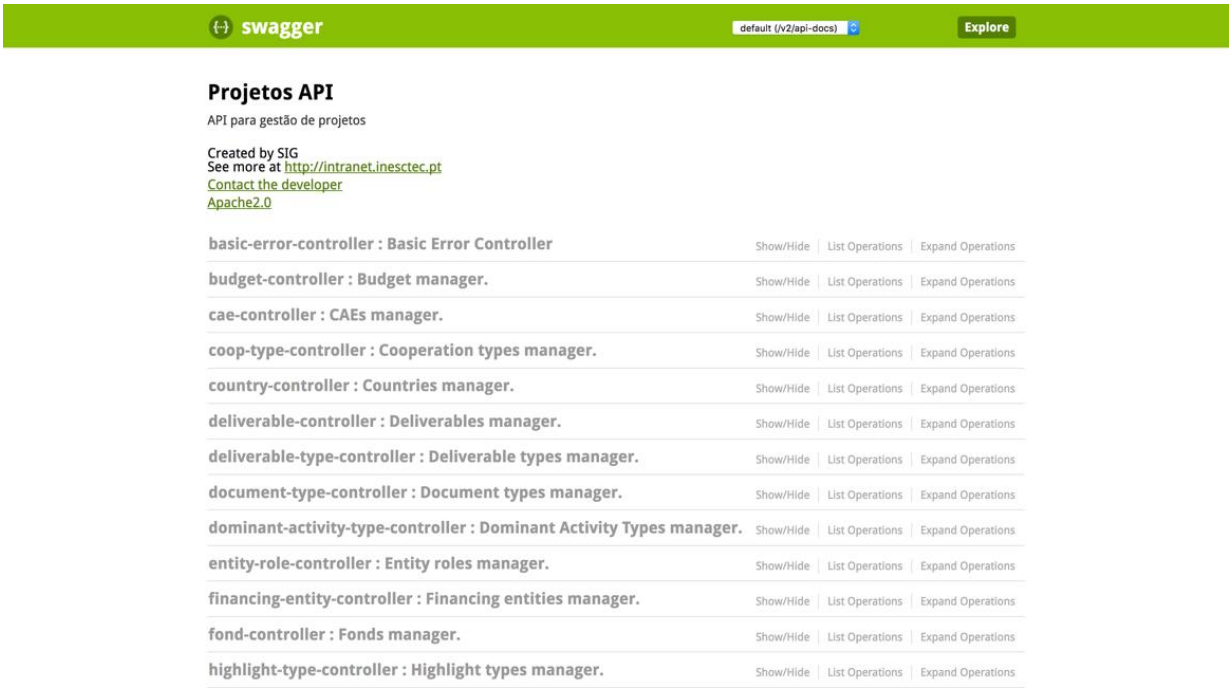


Figure 30 – Online OpenAPI specification of project API

```
{
  “swagger”: “2.0”.
  “info”: { (…) },
  “host”: “iris.inesctec.pt:8081”,
  “basePath”: “/”,
  (…)
  “paths”: {
          (…)
          “/projetoAPI/getProjectById”: {
                “get”: {
                  (…)
                  “summary”: “Get project by id”,
                  (…)
                  “parameters”: [
                    {
                      “name”: “idProject”,
                      (…)
                      “type”: “integer”,
                      (…)
                    }
                  ],
                  “responses”: {
                    “200”: {
                      “description”: “OK”,
                      “schema”: {
                        “$ref”: “#/definitions/ResponseEntity”
                      }
                    },
                    (…)
                  }
                }
              },
          (…)
          “definitions”: {
              “ProjectDTO”: {
                “type”: “object”,
                “properties”: {
                  “beginDate”: {
                    “type”: “string”,
                    “example”: “yyyy-MM-dd HH:mm:ss”
```

```
            },
            “cae”: {
              “type”: “string”
            },
(…)
```

Code 8 – JSON OpenAPI specification of projects API

With this information, it is possible to use a tool called Swagger2GraphQL, that uses the information contained on the OpenAPI specification and turns it into a GraphQL Schema, identifying all the queries and mutations, by the path tag of specification, and types, using the definitions tag. To do this conversion, it must be created a JavaScript project, imported the tool using the NPM[3], added the JSON OpenAPI specification file of projects API, and then executed the command of Code 9.

```
./node_modules/swagger-to-graphql/bin/swagger2graphql –swagger=./projetosAPI.json
> ./projetosAPI.graphqls
```

Code 9 – Command to convert a JSON Open API specification file to a GraphQL Schema

This generation is useful for starting the adoption of GraphQL when there are only REST APIs. However, it generates one GraphQL Schema will all the information, which is a problem when the REST API is extensive. To improve the scalability and organization of the GraphQL Schema, the file was divided into multiple files[4], where which one represents a type with all the queries associated. The schema has the main file, which is the *Project.graphqls* (Code 10) and then other files (Code 11) with the definitions of the types and the extend of the main query type, in order to add the queries available of the correspondent type to the main query defined on *Project.graphqls*.

```
schema {
    query: Query
}

type Project {
    projectId: ID!
    beginDate: String!
    contractReference: String
    (…)
}


# The Root Query for the application
type Query {
    projectsAll: [Project]!
    projectById(projectId: ID!): Project
    projectByOIBase(OIBase: String!): Project
}
```

Code 10 – Main file of GraphQL Schema

```
type FinancingEntity {
    entityId: ID!
    entityAcronym: String
    entityName: String
    entityNameEN: String
    country: Country
```

[3] Packager manager for JavaScript modules
[4] All the GraphQL Schema available at Appendix A – GraphQL Schema

```
    valBegin: String
    valEnd: String
    createUser: String
    createDate: String
    modifyUser: String
    modifyDate: String
}

extend type Query {
    financingEntityById(financingEntityId: ID!): FinancingEntity
}
```

Code 11 – Financing Entity GraphQL Schema

Having the GraphQL Schema defined, it is possible to move to the next steps of the implementation.

Another point in common between the prototypes is the entities definition. The types used during the implementation needed to be transformed into Java classes and respect the rules of the model defined for the project API. As mention on section Prototyping, there is a complex model defined, having the module of the project management at INESC TEC as a base. However, to simplify the study, only some of the entities will be used (Figure 31). This simple version of the model allows us to have all the need information to do a project search by the parameters available on IRIS, such as:

- **Project attributes:** id, short name, OI Base, CAE, begin and end date;
- **Structure attributes:** id and initials;
- **Typology class attributes:** id and acronym;
- **Financing entity attributes:** id and acronym;
- **Institution attributes:** id and short name;
- **Tec4 attributes:** id and short name;
- **Team attributes:** coordinator;
- **Project participation attributes:** idRH.

All the information available with these entities will allow to have distinct outputs.

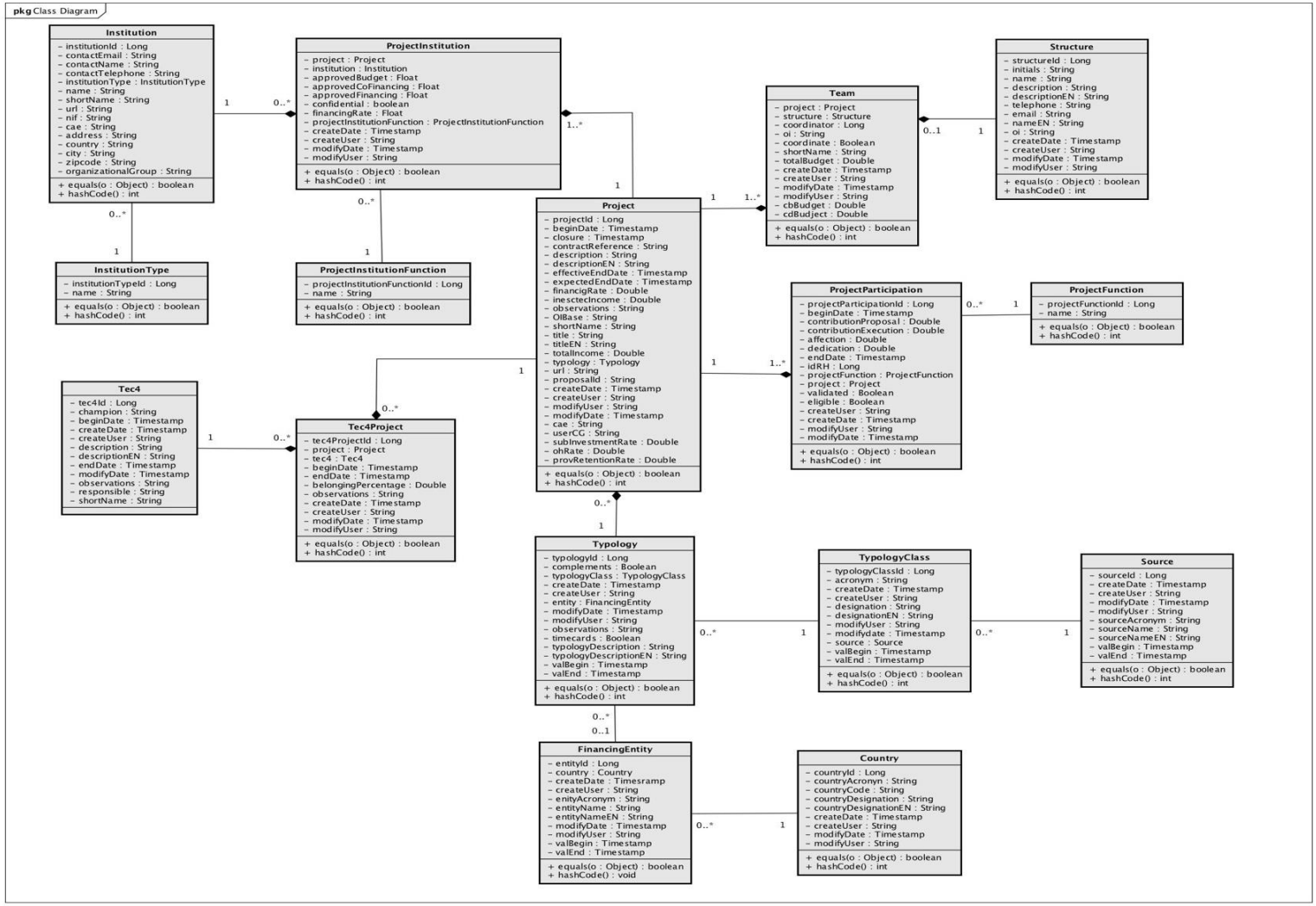


Figure 31 – Entities class diagram

Another component in common is the security one. There is a framework from Spring that allows customizing the authentication and access-control of an application. This framework, called Spring Security, is used on the implementation of the prototypes since it allows to define authentication on specific paths and HTTP methods.

The integration of GraphQL with Spring supplies the API through the path /graphql, so it can be easily defined who can access it. Code 12 is represented the java class responsible for all the management of the API authentication and access control. The *userDetailsService* is responsible for creating the users and associated them with roles. The *configure* method is where is defined who can access determinate paths.

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService() {

      User.UserBuilder users = User.withDefaultPasswordEncoder();
      InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
      manager.createUser(users.username("username").password("password")
      .roles("USER").build());
      return manager;
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
      http.csrf().disable().authorizeRequests()
      .antMatchers("/graphql").access("hasAnyRole('USER')").and()
      .httpBasic().and().sessionManagement()
      .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}
```

Code 12 – Security configuration

The error handler is also shared between the prototypes. To format the GraphQL response in a layout that is composed by the data and the errors information (Code 13), similar to REST API responses, it is needed to specify the error handler, in order to group all the client and server errors in the correct JSON structure, which is represented in the project with the Java class, *GraphQLErrorAdapter*.

```
{
  "data": {
    "projectByOIBase": {
      "projectId": "18809",
      "typology": null,
    }
  },
  "errors": [
    {
      "message": "Typology not found",
      (…)
    }
  ]
}
```

Code 13 - Example of GraphQL API response layout

5.2 Prototype 1 – GraphQL standalone

Prototype 1 represents the full replacement of the REST API for a GraphQL API to data querying. This prototype was implemented following the steps mention on the section API creation. However, the schema was generated from the OpenAPI specification of the REST API to manage the projects, as referred on the section before.

Some libraries[5] are fundamental to the implementation of the prototype:

- **Spring Boot Data Java Persistence API (JPA):** for manipulating the data of the database;
- **Microsoft Java Database Connectivity (JDBC) Driver for SQL Server:** to connect with the database;
- **Spring Boot Security:** for managing the authentication and the access-control;
- **GraphQL Spring Boot UI:** to consult the schema and test requests using an online platform;
- **GraphQL Spring Boot:** for using the integration of Spring Boot with the GraphQL;
- **GraphQL Java Tools:** to aid on mapping the GraphQL Schema into Java objects;
- **Spring Boot Test:** for testing the application.

The resources folder of the solution presents some crucial functions:

- The GraphQL schema files must be inside of it, for Spring use them;
- The integration of Spring with GraphQL also requires the definition of some properties (Code 14) to know the endpoint to GraphQL API and, when it is used, the endpoint for GraphiQL as well;
- The connection to the database is defined on a properties file to locate on this folder.

```
# GraphQL
graphql.servlet.mapping=/graphql
graphql.servlet.enabled=true
graphql.servlet.corsEnabled=true

# GraphiQL
graphiql.mapping=/graphiql
graphiql.endpoint=/graphql
graphiql.enabled=true
graphiql.cdn.enabled=true
graphiql.cdn.version=0.11.11
```

Code 14 - Application properties for GraphQL and GraphiQL of prototype 1

[5] Maven Project Object Model (POM) file available at Appendix C – Prototype 2 POM

On the data layer, it is specifying all the entities need to consult the database, and that has a correspondent type on the GraphQL Schema. It also contains the repositories (Code 15), that are interfaces that manage, between the application and the database, all the CRUD operations, using the *CRUDRepository* of Spring Data, and the custom queries, defined using the tag *@Query*.

```
@Repository
public interface ProjectFunctionRepository extends CrudRepository<ProjectFunction, Long>
{

   @Query(value = "SELECT * FROM ProjectFunction where name = ?1", nativeQuery = true)
   ProjectFunction getProjectFunctionByName(String name);

}
```

Code 15 - Project function repository

On the service layer, it defined the exceptions that the prototype own, to have better feedback when an exception occurs, and the consumer understands the reason. Furthermore, this layer contains the resolvers. The type of resolvers can be separated into two:

- **The main resolver:** where all the queries defined on the schema has their behavior translate into a Java method. This class must implement *GraphQLQueryResolver* in order to do so. Using the prefix get plus the name of the query, automatically there is an association between the method and the query. For this prototype, the behavior implemented is to query the information on the database, using the repositories and then return the information, when it exists, or the exception otherwise;

- **The auxiliary resolvers:** where it is defined how a specific type from the schema will get the value of one attribute which type is another one declared on the schema (Code 16).

```
@Component
public class FinancingEntityResolver implements GraphQLResolver<FinancingEntity> {

    @Autowired
    private CountryRepository countryRepository;


    public Country getCountry(FinancingEntity financingEntity) {
        Optional<Country> result =
        countryRepository.findById(financingEntity.getCountryId());
       if(result.isPresent()){
         return result.get();
       }else{
         throw new CountryNotFoundException("Country not found",
         financingEntity.getCountryId().toString());
       }
     }
}
```

Code 16 - Financing entity resolver of prototype 1

Figure 32 is presented the flow of a data querying request on prototype 1. The consumer makes a POST request with the query to executed and the expected output. That is treated for the *QueryResolver*, which is the main resolver, and the information is search on the database, through the repository. After getting the information, the resolver will check if there is data to

return or not. If there is, even though it is not presented on the sequence diagram, it will go through all the need auxiliary resolvers to obtain all the information need and retrieve it to the consumer. If there is no data to return, then the main resolver returns the exception to the consumer.

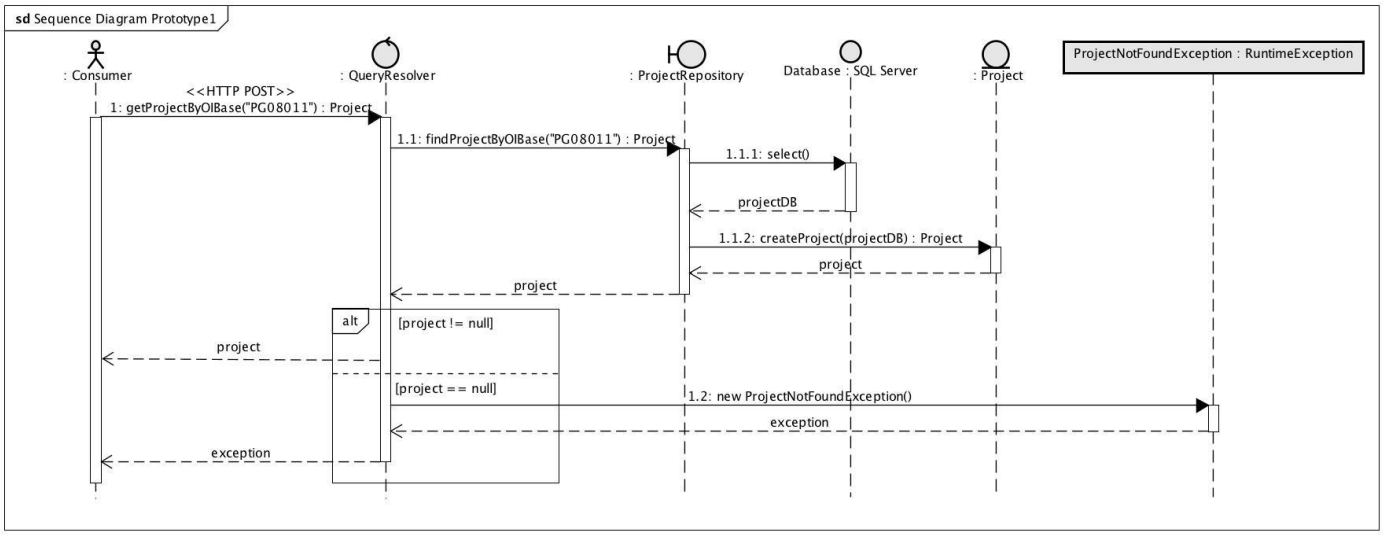


Figure 32 - Sequence diagram of a query in prototype 1

In conclusion, prototype 1 uses the GraphQL directly connected with the database for data querying.

5.3 Prototype 2 – REST with GraphQL

Prototype 2 describes the use of GraphQL to optimize the flexibility of the REST API to data querying. As the prototype mentioned in the section before, this was also implemented following the steps mention on the section API creation and with a schema generated from the OpenAPI specification of the REST API.

The libraries[6] that are fundamental for the implementation of the prototype are the same from prototype 1, except the Spring Boot Data JPA and the Microsoft JDBC Driver for SQL Server, since this API will not connect with the database.

Taking that into consideration, the resources folder of the solution is only constituted by the GraphQL schema files and the integration of the Spring with GraphQL (Code 17).

```
# GraphQL
graphql.servlet.mapping=/graphql
graphql.servlet.enabled=true
graphql.servlet.corsEnabled=true
```

[6] Maven POM file available on Appendix C – Prototype 2 POM

```
# GraphiQL
graphiql.mapping=/graphiql
graphiql.endpoint=/graphql
graphiql.enabled=true
graphiql.cdn.enabled=true
graphiql.cdn.version=0.11.11
```

Code 17 - Application properties for GraphQL and GraphiQL of prototype 2

On the data layer, it is only specified the entities that have an equivalent type on the GraphQL Schema.

On the service layer, it is defined as the exceptions that the prototype own, as the prototype 1. Furthermore, this layer contains the resolvers. The type of resolvers can be separate in two, like the prototype described on the section before, but with distinct behavior:

- **The main resolver:** for this prototype, the behavior implemented is to query the information on the REST API, using the HTTP requests and then return the information, when exists, or the exception otherwise (Code 18);

```
public Project getProjectById(Long projectId) {
    HttpHeaders headers = createHeaders("username","password");

    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<String> entity = new HttpEntity<String>("parameters", headers);

    UriComponentsBuilder builder = UriComponentsBuilder
    .fromHttpUrl("https://iris.inesctec.pt/projetoAPI/getProjectById");

    builder.queryParam("idProject", projectId);

    ResponseEntity<Project> result = restTemplate.exchange(builder.build().encode()
    .toUri(), HttpMethod.GET, entity,Project.class);

    Project project = result.getBody();
    if(Objects.nonNull(project)){
        return project;
    }else{
        throw new ProjectNotFoundException("Project not found", projectId.toString());
    }
}
```

Code 18 - Example of a method from a query resolver

- **The auxiliary resolvers:** with the same responsibility defined on the prototype 1, but the query is made using a REST web service (Code 19).

```
@Component
public class FinancingEntityResolver implements GraphQLResolver<FinancingEntity> {

    public Country getCountry(FinancingEntity financingEntity) {
          HttpHeaders headers = createHeaders("username","password");

          headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
          HttpEntity<String> entity = new HttpEntity<String>("parameters", headers);

          UriComponentsBuilder builder = UriComponentsBuilder
          .fromHttpUrl("https://iris.inesctec.pt/projetoAPI/getCountryById");

          builder.queryParam("idCountry", financingEntity.getCountryId());

          ResponseEntity<Country> result = restTemplate.exchange(builder.build().encode()
```

```
                .toUri(), HttpMethod.GET, entity,Country.class);

                Country country = result.getBody();
                if(Objects.nonNull(country)){
                        return country;
                }else{
                        throw new CountryNotFoundException("Country not found",
                        financingEntity.getCountryId().toString());
                }
        }
}
```

Code 19 - Financing entity resolver of prototype 2

The flow of a data querying request on prototype 2 is presented at Figure 33. The consumer makes a POST request with the query to executed and the expected output, like the prototype 1. That request is treated for the QueryResolver, which is the main resolver, and the information is search on the REST API, through an HTTP request. Having the information, the QueryResolver checks if there is data to return or not. If there is nothing to return, then the main resolver returns the exception to the consumer. If there is data, even though is not presented on the sequence diagram, the application will go through all the need auxiliary resolvers to obtain all the information need, making more requests to the REST API, and retrieve it to the consumer.

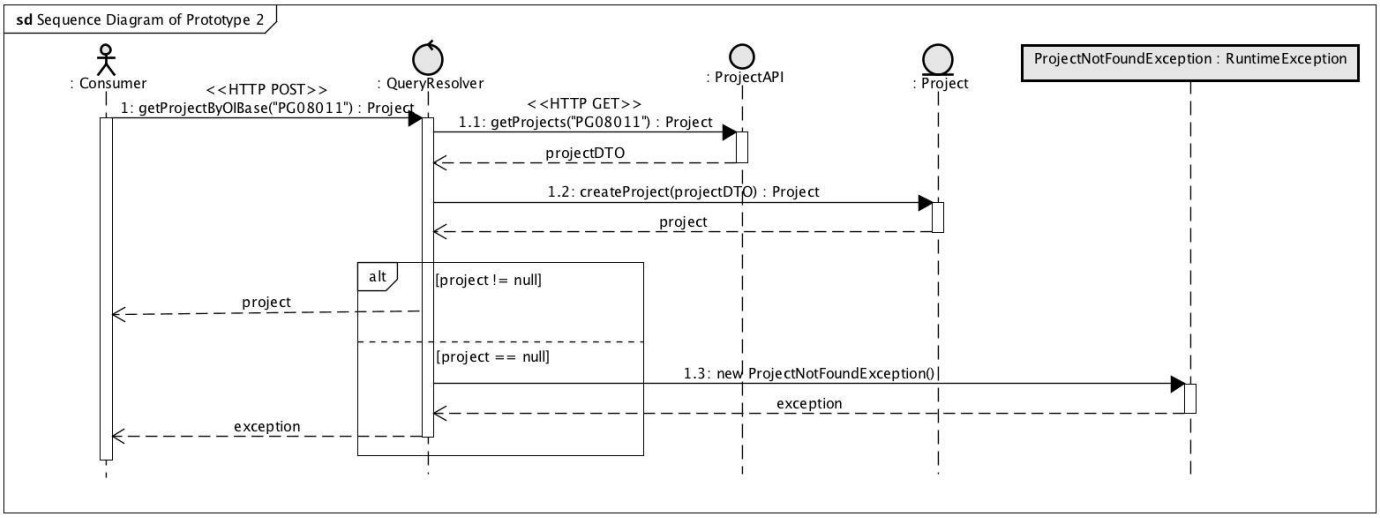


Figure 33 - Sequence diagram of a query in prototype 2

In conclusion, prototype 2 uses the GraphQL to auxiliary the flexibility of the REST API, which can bring improvements on the size of the request.

6 Tests and Solution Evaluation

Have no fear of perfection, you will never reach it.
(Salvador Dali)

The experimentation and evaluation have as main goal to validate if the solution presented in this thesis resolves the problem presented. For this, it is necessary not only to test the final solution, but also its implementation.

In order to validate the proposed solution, it is needed to define which measurements will be evaluated, hypotheses will be tested and how this process will be made.

In this chapter, it is presented all the testing and solution evaluation and justified all the conclusions from the obtained results.

6.1 Measurements to Evaluate

The measurements to evaluate allows defining which concepts are necessary to be taken into consideration when a solution is being evaluated. There are many measurements, but it should be chosen the one that is related to the proposed solution and its requirements.
For validate the proposed solution, it is required to take into consideration the following measurements, based on the requirements presented on Requirements Analysis and the problem explained on Problem:

- **Technical quality (accuracy):** all the solutions must be tested, in order to discard the influence of bad implementation on the result analysis;

- **Performance:** the primary goal of this thesis is to find a solution that presents a better performance, so this is a crucial measure, which will be evaluated observing the next measures, taking into account tests with distinct complexities:

- **Time:** the response times of the API must be improved, thus studying them, it can be found the one solution that presents better results;
- **Size:** the response size can also affect performance since it can exist solutions that deal better with short and straightforward API's responses than with more complex and bigger ones.

These measures will help on evaluating and analyzing the different solutions presented on this project, allowing to define the one that is the best response to the problem and the advantages and disadvantages of each one.

6.2 Hypotheses and Evaluation Methodology

The experimentation process had an experimental life cycle (Figure 34) that started with an exploratory analysis to observe the data (observations of the environment). This analysis was made during the problem characterization (Problem) and the studying of state of the art (State of the Art). After that, the hypotheses were constructed, followed by the experimentation phase with the data analysis and the attempt to obtain conclusions (Gomes, 2018).

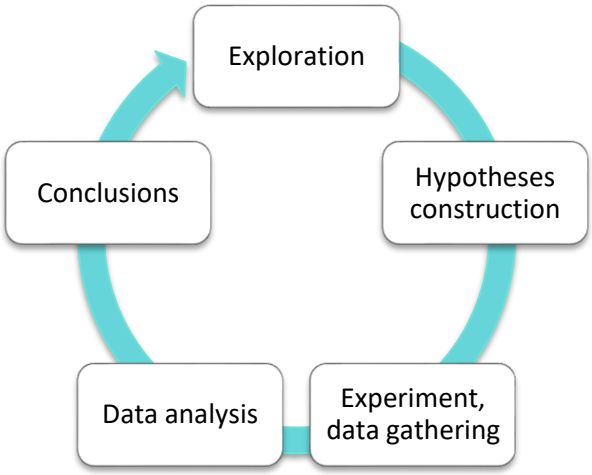


Figure 34 - Experimental life cycle, adapted from (Gomes, 2018)

6.2.1 Hypotheses

The hypotheses formulation for this project was made after the definition of the problem and the exploration of state of the art. The hypothesis is a valid assumption that can be made by the exploration of a set of data, and it should be validated. During its construction, it is defined as two types of hypotheses (Manso, 2003):

- **H_0:** null hypothesis, usually conservative;

- **H1:** alternative hypotheses.

The specify hypotheses for this thesis are the following ones:

- **H_{01}:** REST API presents the same performance as a GraphQL API;
- **H_{02}:** REST API has the same performance as a solution that uses REST with GraphQL;
- **H1:** REST API demonstrates better performance in atomic calls than GraphQL API;
- **H2:** GraphQL API offers better performance than REST API when multiple endpoints are consulted;
- **H3:** The integration of GraphQL with a REST API shows better performance results than a REST API standalone.

The hypotheses **H_{01}** and **H_{02}** are conservative, so it is considered that it will not be any differences between the solutions.

The **H1** is formulated, taking into consideration the study presented by Guillen-Drija, described on Comparison between REST and GraphQL. This hypothesis recognizes that GraphQL API will not have better performance than REST API when a simple request is made.

Like the hypothesis **H1**, **H2** is also constructed looking for the results of the experiment of Vázquez-Ingelmo, demonstrated in the same section. The **H2** complements analyses the performance on complex requests, where GraphQL should present better performance.

Lastly, H3 is a hypothesis that derived from the implementation of GraphQL with REST API made by Netflix, introduced on Adoptions overview. This hypothesis is the reason why an architecture of prototype 2 (Prototype 2 – REST with GraphQL) was considered.

6.2.2 Evaluation Methodology

In order to evaluate the hypotheses presented in the previous section and considering the measurements defined (Measurements to Evaluate), it must be defined as the methodology used. All the solutions described in this thesis will be tested on IRIS's environment.

The technical quality of each solution was debugged by the typical software engineering tests like unit tests and integration (JUnit), code coverage (JaCoCo), loading tests (Apache JMeter), and acceptance tests.

With technical quality ensured, it is possible to test the performance and validated the hypotheses formulated. For the time and size, it was used a tool called PRTG Network Monitor, that allows sending a distinct request to the APIs, with a defined interval. The analysis of these two measures permits to test the behavior of each solution for these measures. Those metrics

were evaluated, taking into consideration requests with distinct complexities. The request complexity was classified, taking into consideration the number of endpoints, on the case of the REST API, or types, on GraphQL API, that has to be consumed in order to obtain all the necessary data. Table 18 presents the classification of complexity that will be used during the tests.

Table 18 - Classification of different levels of complexity

Complexity Classification	Number of Endpoints/Entities to Achieve all the Information
Low	1
Normal	Between 2 and 4
High	More than 5

After collecting all the results from each measure, statistical tests were made. The statistical tests grant to take a decision, using the data observed on specific experimentation. There are parametric and non-parametric tests, so it is necessary to analyze, for example, the results numbers, if they are quantitative or qualitative or if they are independent or not.

For this project, the solutions are not executed together. Thus, it can be classified as independent. The same measures were tested on different samples so that it can be used the parametric test Analysis of Variance (ANOVA) or the non-parametric tests: Kruskal-Wallis test or Mood's test (XLSTAT, 2019). Since the tests were made based on some distribution, it was used ANOVA to test the hypotheses. Other benefits that ANOVA tests have been the better controlling of type 1 error and allow to compare multiple groups.

6.3 Experimentation and Evaluation

Following the evaluation methodology defined on Evaluation Methodology, it is necessary to test and evaluate the prototypes created and compare them with the current solution. This evaluation will allow concluding which one represents a better solution for the problem demonstrated on subsection Problem.

6.3.1 Technical quality

The first step in experimentation is to test the technical quality of the two prototypes. It is vital to ensure that the results of each solution are not affected by bad development.

The technical quality is established by unit and integration tests, code coverage, loading tests, and acceptance tests. The core of the two prototypes is similar, so the unit testing is identical as well. The main unit tests apply to the solutions involve the resolvers and test the following aspects:

- Return lists of data with the correct dimension;

- Return lists or objects with the right data for each attribute;
- Return the custom exception, when the information is not available;
- Validation of methods inputs (null, empty).

To keep the tests valid through time, on the unit testing of each resolver, the information expected is simulate using mock objects. On prototype 1, the return from the methods of the repository are simulated and, on prototype 2, the REST API responses are mocked. On Code 20, an example of a unit test used on prototype 1 is presented, to test the throw of a custom exception.

```
@Test(expected = ProjectNotFoundException.class)
public void whenFindProjectById_thenReturnProjectNotFoundException() {
    // given
    Optional<Project> projectOptional = Optional.empty();

    when(projectRepository.findById(new Long(1))).thenReturn(projectOptional);

    // then
    queryResolver.getProjectById(new Long(1));
}
```

Code 20 - Example of a unit test of prototype 1

The other tests that use JUnit are the integration tests, that test the possibility of integration of the API with other systems, like a consumer, database, and REST API. The relation with API consumer is equal on the two prototypes, so the tests are equal. In these tests are tested the status of API response and the response format, example on Code 21. Prototype 1 is tested the integration of the GraphQL API with the SQL Server database. Prototype 2 is tested the integration of the GraphQL API with the REST API.

```
@Test
public void testGetProjectByProjectId() throws Exception {
    String query = "query{\n" +
            "  projectById(projectId:\"18809\"){\n" +
            "    projectId\n" +
            "    title\n" +
            "    shortName\n" +
            "    OIBase\n" +
            "    typology{\n" +
            "      typologyDescription\n" +
            "    }\n" +
            "    teams{\n" +
            "      coordinator\n" +
            "      oi\n" +
            "      shortName\n" +
            "      totalBudget\n" +
            "    }\n" +
            "  }\n" +
            "}";

    GraphQLQuery graphlqlQuery = new GraphQLQuery();
    graphlqlQuery.setQuery(query);

    HttpHeaders headers = createHeaders("username", "password");
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));

    HttpEntity<Object> entity = new HttpEntity<Object>(graphlqlQuery, headers);
```

```
        ResponseEntity<String> results =
            restTemplate.exchange("http://localhost:8080/graphql",
                HttpMethod.POST, entity, String.class);

        assertEquals(results.getStatusCode(),HttpStatus.OK);

    }
```

Code 21 - Example of an integration test of prototype 1 and 2

All the tests are passing correctly (Figure 35) and, using the JaCoCo, the code coverage of the functional code [7] is around 90% on the two prototypes (Figure 36). Since entities and repositories do not contain functional code, they are excluded from the code coverage.

Figure 35 - Results of unit testing

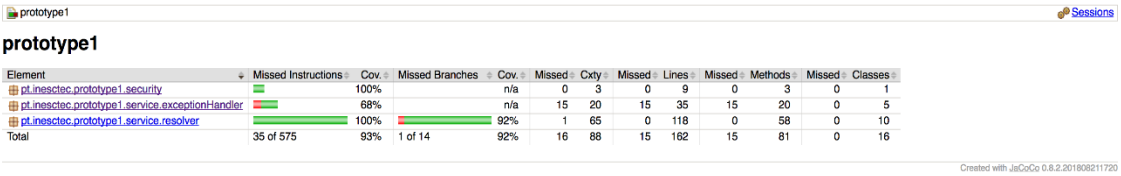


Figure 36 - Code coverage results

Even though the guarantee of the correct functional behavior of the prototypes and a good percentage of code tested, some non-functional requirements must be supported too.

JMeter is the tool used to make the load tests. For each query available in the prototypes, it was created a load test. Each test considers a concurrent number of requests, define as 550 since is the mean of active users of IRIS and it cannot has a response size more prominent than 53 kilobytes and a response time bigger than 20 seconds, which are some values similar to REST API responses (see INESC TEC). There is also a data JSON assertion to verify the correct format of responses. The result of the load test is 100% correct for the two prototypes (Figure 37).

[7] Code with custom behavior. For example, getters and setters are not included.

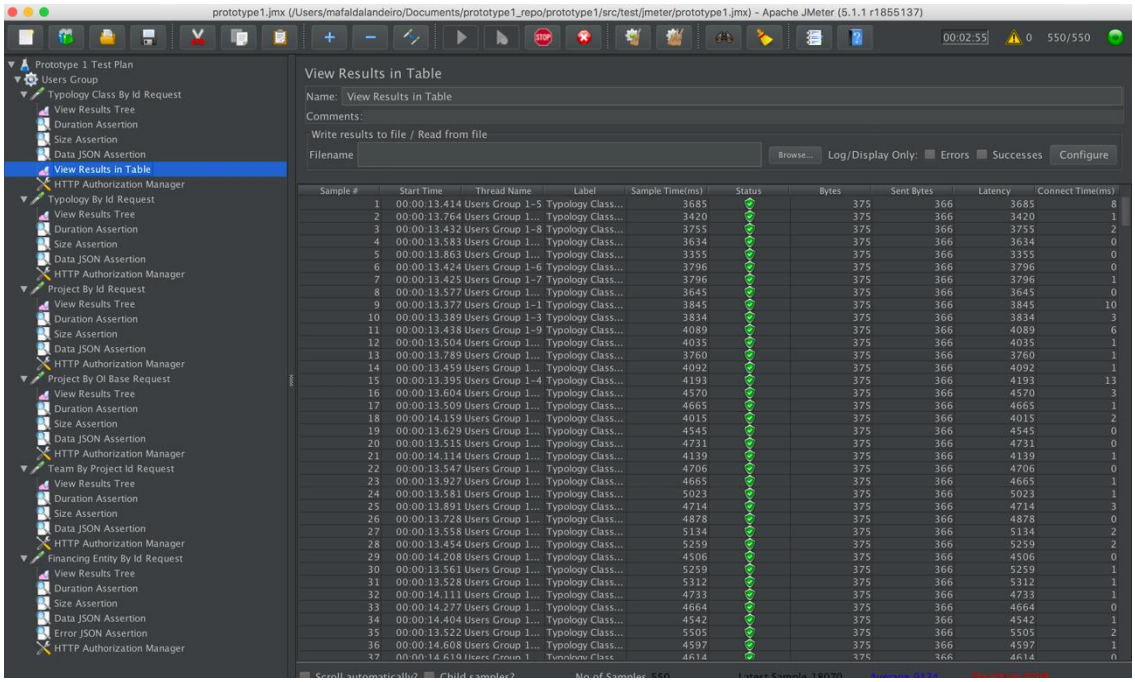


Figure 37 - Load testing results

Finally, using the GraphiQL interface, some acceptance tests[8] were manually made, in order to validate the correct behavior of the API (example in Table 19).

Table 19 - Example of acceptance test

Data	
Test Number	1
Test Goal	Validate the result from searching for a project using OI.
Input	Search OI “PG08011” and expected project id, title, and short name as a result.
Expected output	A JSON object with project id, title, and short name.
Real Results	
Prototype 1	A JSON object with project id, title, and short name.
Prototype 2	A JSON object with project id, title, and short name.

It can conclude that the prototypes have technical quality since they have success in functional and non-functional tests.

6.3.2 Performance

After measuring and evaluate the technical quality of the prototypes, it is needed to test the performance. The performance was measured using distinct request complexities, as mentioned on Evaluation Methodology, and it is evaluated using metrics of time and size. In order to obtain the information need, it was used the PRTG tool, which allows creating sensors

[8] Consult acceptance tests on Appendix D – Acceptance tests

(Figure 38). These sensors have a specific configuration[9] associated that makes a request to an API from time to time and registers the response time and the response size.

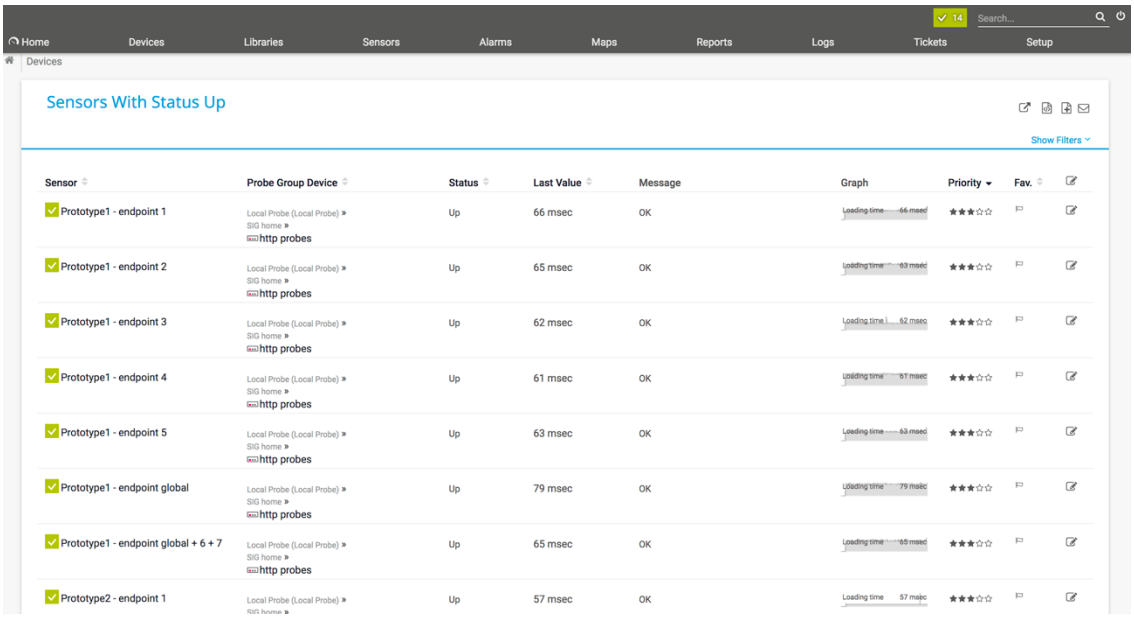


Figure 38 - Sensors view from PRGT

To create a table with a sample of results, using distinct complexities, it was used the following query requests to the GraphQL APIs (prototype 1 and 2):

- Complexity low (Code 22)

```
query {
  financingEntityById(financingEntityId:4) {
          entityName
          entityAcronym
  }
}
```

Code 22 - Query request for complexity low

- Complexity normal (Code 23)

```
query {
  projectByOIBase(OIBase:”PG08011”) {
          OIBase
          shortName
          typology {
                  typologyId
          }
          beginDate
          expectedEndDate
          title
          description
          proposalId
  }
}
```

[9] Consult example of configuration file on Appendix E – Example of a sensor configuration

Code 23 - Query request for complexity normal

- Complexity high (Code 24)

```
query {
  projectByOIBase(OIBase:”PG08011”) {
          OIBase
          beginDate
          expectedEndDate
          contractReference
          teams {
                  coordinator
                  oi
                  coordinate
                  structure {
                          sigla
                  }
          }
          typology {
                  typologyDescription
                  typologyClass {
                          designation
                  }
                  financingEntity {
                          entityAcronym
                  }
          }
  }
}
```

Code 24 - Query request for complexity high

PGRT generates a report with the necessary information and using the data mention on the subsection INESC TEC for the current solution; it generates the Table 20, Table 21 and Table 22, with the mean of the requests time and size for each API, accordingly with the request complexity.

- Complexity low (Table 20)

Table 20 - Mean results for requests with complexity low

	Time (ms)	**Size (kB)**
Current solution	129	0,66
Prototype 1	203	0,13
Prototype 2	195	0,07

- Complexity normal (Table 21)

Table 21 - Mean results for requests with complexity normal

	Time (ms)	**Size (kB)**
Current solution	4236	6,58
Prototype 1	255	0,98
Prototype 2	480	1,15

- Complexity high (Table 22)

Table 22 - Mean results for requests with complexity high

	Time (ms)	Size (kB)
Current solution	13480	28,97
Prototype 1	2070	0,77
Prototype 2	830	1,51

With this information, it is possible to observe that any API do not share the same response time or size in the distinct complexities' cases. Observing Figure 39, the prototypes generally present better response time than the current solution, except on complexity low, where prototype 1 has the worst result than the others. On Figure 40, it can be concluded that the prototypes improve the response size significantly comparing with the current solution.

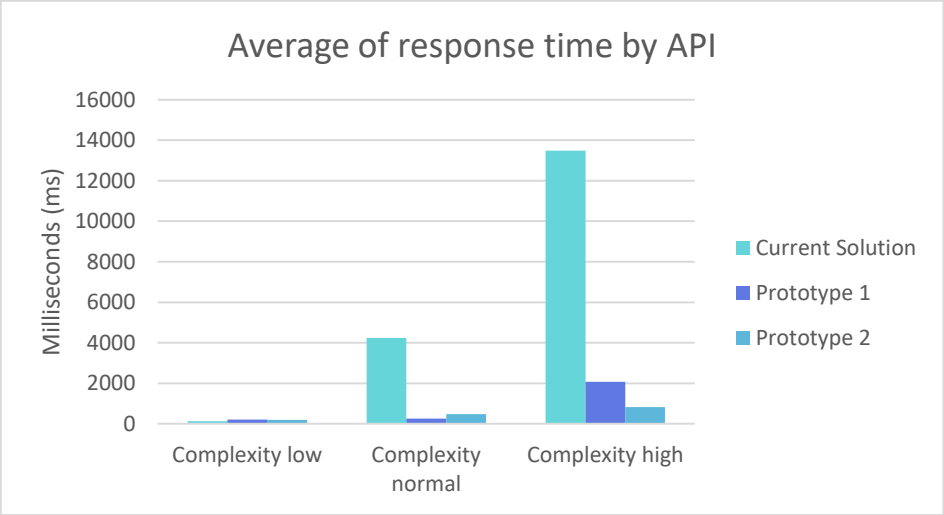


Figure 39 - Average of response time by API

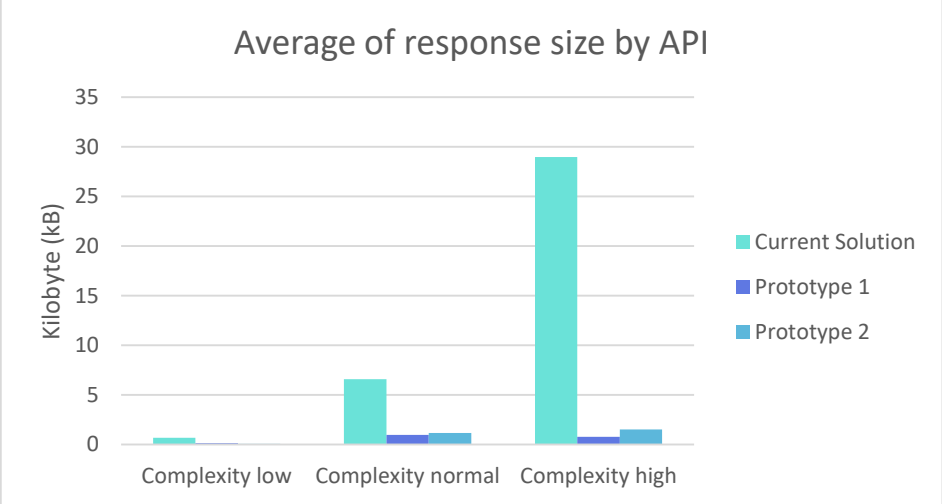


Figure 40 - Average of response size by API

Using the experimentation results and applying the ANOVA testing[10], it is possible to classify as accurate of false the hypotheses specify on subsection Hypotheses:

- **H_{01}:** REST API presents the same performance as a GraphQL API;

  This hypothesis is false since neither the response time or the response size are similar between the current solution and the prototype 1. Comparing each other, with a confidence interval of 95% and the Tukey's Honestly Significantly Different (HSD) test, they are significantly distinct in response time and response size (Table 23).

Table 23 – Differences' analysis between the current solution and prototype 1 (Tukey's HSD test)

Measure	Difference	Standardized difference	Critical value	Pr > Diff	Significant	Lower bound (95%)	Upper bound (95%)
Time	5710	4	2	0	Yes		|||||||||||||||
Size	11	3	2	0	Yes		|||||||||||||||

- **H_{02}:** REST API has the same performance as a solution that uses REST with GraphQL;

  This hypothesis is also false since neither the response time or the response size are similar between the current solution and prototype 2. Comparing each other, with the test mention before, they are significantly distinct in response time and response size (Table 24).

Table 24 – Differences' analysis between the current solution and prototype 2 (Tukey's HSD test)

Measure	Difference	Standardized difference	Critical value	Pr > Diff	Significant	Lower bound (95%)	Upper bound (95%)
Time	5520	3	2	0	Yes		|||||||||||||||
Size	11	3	2	0	Yes		|||||||||||||||

- **H1:** REST API demonstrates better performance in atomic calls than GraphQL API;

  The hypothesis can only be considered valid if we give more significant weight to the time measure. The response time is better on the REST API; however, the response size is better in GraphQL API. This can be concluded observing the mean results on experimentation with complexity low, presented before.

- **H2:** GraphQL API offers better performance than REST API when multiple endpoints are consulted;

[10] See descriptive statistics in Appendix F – Descriptive statistics (quantitative data) for time and Appendix G – Descriptive statistics (quantitative data) for size

This hypothesis is correct because it presents better response time and size on requests with complexity normal and high, which are the ones that involve more than one endpoint/entity. On complexity normal, the GraphQL API is 93% better at the response time and 85% better at the response size. Looking at the complexity high, prototype 1 is 84% better at the response time and 97% better at the response size.

- **H3:** The integration of GraphQL with a REST API shows better performance results than a REST API standalone.

  Like the **H1**, this condition can have distinct classifications, accordingly with the weight given to the time and size measures. On normal and high complexities, prototype 2 is better. However, on low complexities, the response time is better 33%.

Taking into consideration that, globally the prototypes present better performance than the current solution, the comparison between them is hard, since they are not significantly different. The mean of response size is equal. However, to give a response about a possible implementation to solve the performance issue of IRIS, the result of ANOVA testing is clear: the prototype 1 presents the best performance (Figure 41), considering all the complexities.

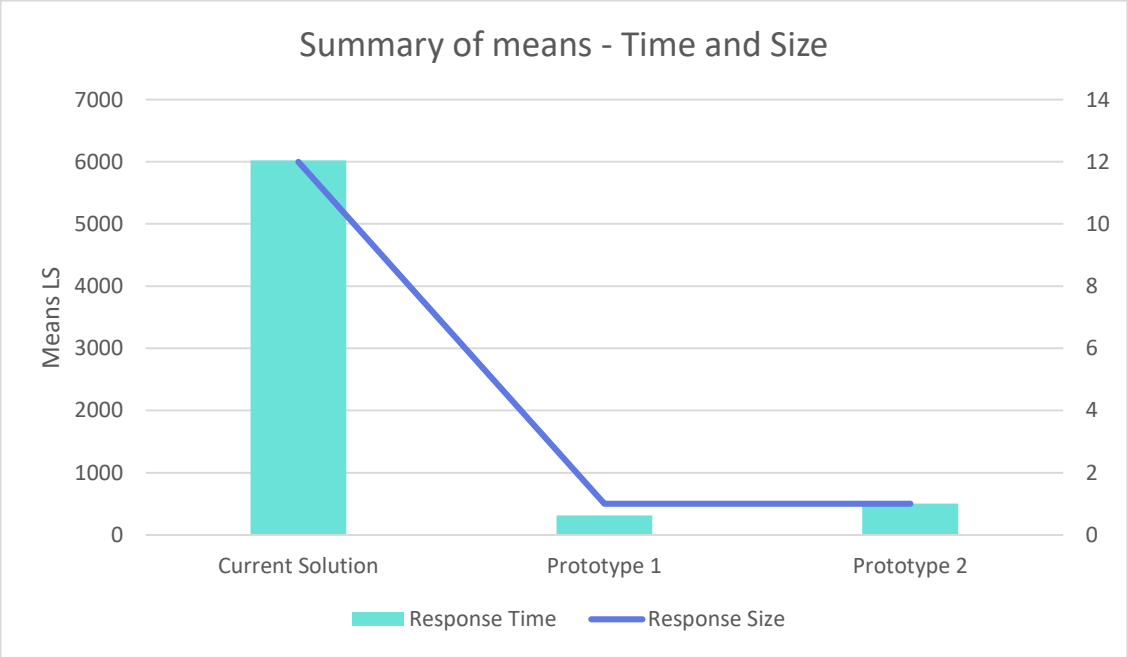


Figure 41 - Summary of means - time and size

7 Conclusion

It is fine to celebrate success, but it is more important to heed the lessons of failure.
(Bill Gates)

This section summarizes what it was done, difficulties found, but also future work and contributions.

7.1 Work Done and Difficulties Found

This document presents all the steps made in order to find a solution for a performance issue related to a REST API. All these steps were made accordingly with the methodology defined in the subsection Methodology.

Firstly, it was analyzed the problem that the INESC TEC have on a platform responsible for the management of data, called IRIS. The problem was apparent: the response time of REST API was not right and attached to that, the response size was bigger than need as well. Then, a process to find a solution was followed, which is described on the subsection Business and Innovation Process. There were three possible solutions on the table, and the decision was to take a step into GraphQL.

This technology is new to the stack of the team, and it has also involved some time in learning how it works, and the best practices associated with it. Due to the limit time to present the thesis, the learning curve had to be short, and it was possible since GraphQL is simple and easy to learn.

All the learning helped on building the State of the Art, mainly on the GraphQL description. The description of REST and GraphQL helped on understanding the core principles of each one and how one could improve the other. Besides that, there are some documented adoptions of GraphQL made by known companies like Netflix, GitHub, and PayPal, which help to understand the process that a possible transformation from REST to GraphQL would require.

One of the main difficulties during this thesis was the lack of scientific papers related with the subject, and the way to work around that was to explore grey documentation and be always attentive to the new papers that could be published during the execution of the thesis.

Although the difficulties, it was created state of the art with enough information to consider distinct design alternatives and build some hypotheses to validate.

Another difficulty was the extension of the work related to the designs that could be implemented. The most two promising prototypes were implemented and with good technical quality (tests presented on section Technical quality). The two prototypes use GraphQL, and they reveal to be, on the overall, better in terms of performance than the current solution.

The research question of this thesis was:

> *Can GraphQL improve searching performance, while also providing flexibility, when compared to the existing solution?*

Even though the flexibility was not taking into consideration for testing, the GraphQL presents better flexibility on API requests, a fact that has been proved, for example, on the study made by Stubailo. For the tests made, only one endpoint was requested for all the GraphQL APIs with no more data than needed. Giving a developer perspective, there is no need to worry about what each consumer may want; all the information is available, so they manage by themselves their necessities. However, the question has an answer now: yes, GraphQL can improve searching performance, considering the measures time and size, as it is proved on the chapter of Tests and Solution Evaluation.

In conclusion, the analysis of GraphQL performance brought a response to the integration of GraphQL with REST and the adoption of GraphQL, highlighting the benefits that this technology can bring in terms of performance to APIs.

7.2 Future Work

GraphQL is a recent technology, so there are many details to explore. Related to the thesis, the analyze of performance should be tested on more APIs, and the alternative of design not implemented should be explored too. The evaluation proved that REST APIs have a better response time in atomic calls than GraphQL APIs, so the alternative could be an integrative solution that can bring together the best of each solution.

The performance was studied using specific technologies (Java and Spring Boot) and, in order to validate the overall performance, should be discarded the influence of technologies used to implement the APIs. It is also necessary to explore performance with distinct types of complexities. In this study, the complexity was related to the endpoints/entities to query; however, it can also be explored request sizes, for example.

To sum up, this is one more step towards the impact of using GraphQL could bring. However, there is a lot to explore still.

7.3 Contributions

The context of this thesis was applied to INESC TEC, mainly on the IRIS platform. Since it was a new technology, the team acquired some knowledge about it and learned how it could contribute to the implementation of APIs.

After this thesis, the prototype selected as the one that presented the best performance was not applied on production yet. However, it will be beneficial for a diversity of functionalities that are planned to be implemented and, since GraphQL can be used by distinct programming languages, this alternative can be also considered to others platforms that struggle with the same problem.

References

baeldung (2017) *Getting Started with GraphQL and Spring Boot* [Online]. Available at https://www.baeldung.com/spring-graphql (Accessed 15 May 2019).

Brito, G., Mombach, T. and Marco Tulio Valente (2019) 'Migrating to GraphQL: A Practical Assessment', *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, IEEE, pp. 140–150 [Online]. DOI: 10.1109/SANER.2019.8667986 (Accessed 30 May 2019).

Facebook Inc. (2018) *GraphQL: A query language for APIs.* [Online]. Available at http://graphql.org/ (Accessed 17 October 2018).

Fielding, R. T. and Taylor, R. N. (2000) 'Principled design of the modern Web architecture', *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pp. 407–416 [Online]. DOI: 10.1145/337180.337228 (Accessed 31 October 2018).

Gomes, E. F. (2018) 'Experimental design', Moodle ISEP [Online]. Available at https://moodle.isep.ipp.pt/mod/resource/view.php?id=139142 (Accessed 1 February 2019).

Guillen-Drija, C., Quintero, R. and Kleiman, A. (2018) *GraphQL vs REST: una comparación desde la perspectiva de eficiencia de desempeño.*, [Online]. DOI: 10.13140/RG.2.2.25221.19680 (Accessed 1 December 2018).

INESC TEC (2017) *INESC TEC* [Online]. Available at https://www.inesctec.pt/en/institution (Accessed 22 February 2019).

Krivtsov, R. (2016) 'Moving existing API from REST to GraphQL', *Open GraphQL* [Online]. Available at https://medium.com/open-graphql/moving-existing-api-from-rest-to-graphql-205bab22c184 (Accessed 20 January 2019).

Manso, E. (2003) 'Experimentation in Software Engineering', Oporto [Online]. Available at https://www.infor.uva.es/~manso/disexperoporto-web.pdf (Accessed 1 February 2019).

Marek, A. and Baker, B. (2019) *GraphQL Java* [Online]. Available at https://www.graphql-java.com/ (Accessed 20 January 2019).

Möller, S. (2010) 'Usability Engineering', in Möller, S. (ed), *Quality Engineering: Qualität kommunikationstechnischer Systeme*, Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 57–74 [Online]. DOI: 10.1007/978-3-642-11548-6_4 (Accessed 10 October 2018).

Nicola, S., Ferreira, E. P. and Ferreira, J. J. P. (2012) 'A novel framework for modeling value for the customer, an essay on negotiation', *International Journal of Information Technology & Decision Making*, vol. 11, no. 03, pp. 661–703 [Online]. DOI: 10.1142/S0219622012500162.

Nordic APIs (2017) *eBook Released: GraphQL or Bust | Nordic APIs |* [Online]. Available at https://nordicapis.com/ebook-released-graphql-bust/ (Accessed 17 October 2018).

Open API Initiative (2017) *OpenAPI* [Online]. Available at https://www.openapis.org/ (Accessed 26 December 2017).

Osterwalder, A. and Pigneur, Y. (2011) *Business Model Generation: Inovação em Modelo de Negócios*, 1ª., Rio de Janeiro, Alta Books [Online]. (Accessed 1 February 2019).

Pivotal (2019) *Spring* [Online]. Available at https://spring.io/ (Accessed 20 January 2019).

Porcello, E. and Banks, A. (2018) *Learning GraphQL - Declarative Data Fetching for Modern Web Apps*, 1st edn, O'Reilly [Online]. Available at http://shop.oreilly.com/product/0636920137269.do (Accessed 25 November 2018).

Rettig, M. (1994) 'Prototyping for Tiny Fingers', *Commun. ACM*, vol. 37, no. 4, pp. 21–27 [Online]. DOI: 10.1145/175276.175288.

Saaty, T. L. and Vargas, L. G. (1991) *Prediction, Projection and Forecasting: Applications of the Analytic Hierarchy Process in Economics, Finance, Politics, Games and Sports*, Springer Netherlands [Online]. Available at https://www.springer.com/kr/book/9789401579544 (Accessed 1 February 2019).

Shtatnov, A. and Ranganathan, R. S. (2018) 'Our learnings from adopting GraphQL', *Netflix TechBlog* [Online]. Available at https://medium.com/netflix-techblog/our-learnings-from-adopting-graphql-f099de39ae5f (Accessed 9 February 2019).

Stuart, M. (2018) 'GraphQL: A success story for PayPal Checkout', *PayPal Engineering* [Online]. Available at https://medium.com/paypal-engineering/graphql-a-success-story-for-paypal-checkout-3482f724fb53 (Accessed 17 January 2019).

Stubailo, S. (2017) *GraphQL vs. REST* [Online]. Available at https://blog.apollographql.com/graphql-vs-rest-5d425123e34b (Accessed 3 December 2018).

Sturgeon, P. (2017) *GraphQL vs REST: Overview | Phil Sturgeon* [Online]. Available at https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/ (Accessed 17 October

2018).

Terzić, B., Dimitrieski, V., Kordić (Aleksić, S., Milosavljevic, G. and Luković, I. (2017) 'MicroBuilder: A Model-driven Tool for the Specification of REST Microservice Architectures', *Int. Conf. on Information Society and Technology* [Online]. DOI: https://doi.org/10.1080/17517575.2018.1460766 (Accessed 10 January 2019).

Torikian, G., Black, B., Swinnerton, B., Somerville, C., Celis, D. and Daigle, K. (2016) *The GitHub GraphQL API* [Online]. Available at https://githubengineering.com/the-github-graphql-api/ (Accessed 17 January 2019).

Vacek, J. (2012) 'Innovation management', Technology, UWB, Faculty of Economics [Online]. Available at https://www.slideshare.net/amibiriba/innovation-management-11735406 (Accessed 1 February 2018).

Vázquez-Ingelmo, A., Cruz-Benito, J. and García-Peñalvo, F. J. (2017) 'Improving the OEEU's Data-driven Technological Ecosystem's Interoperability with GraphQL', *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM 2017, New York, NY, USA, ACM, pp. 89:1–89:8 [Online]. DOI: 10.1145/3144826.3145437 (Accessed 17 October 2018).

Wiegers, K. and Beatty, J. (2013) *Software Requirements*, 3rd edn, Redmond, Washington, Microsoft Press [Online]. (Accessed 20 January 2019).

Wittern, E., Cha, A. and Laredo, J. A. (2018) 'Generating GraphQL-Wrappers for REST(-like) APIs', Mikkonen, T., Klamma, R., and Hernández, J. (eds), *Web Engineering*, Lecture Notes in Computer Science, Springer International Publishing, pp. 65–83 [Online]. (Accessed 20 January 2019).

XLSTAT (2019) *Which statistical test should you use?* [Online]. Available at https://help.xlstat.com/customer/en/portal/articles/2062457-which-statistical-test-should-you-use-?b_id=9202 (Accessed 17 February 2019).

Appendixes

Appendix A – GraphQL Schema

```
schema {
    query: Query
}

type Project {
    projectId: ID!
    beginDate: String!
    contractReference: String
    description: String
    descriptionEN: String
    effectiveEndDate: String
    expectedEndDate: String!
    financingRate: Float
    subInvestmentRate: Float
    ohRate: Float
    provRetentionRate: Float
    inesctecIncome: Float
    observations: String
    OIBase: String!
    shortName: String!
    title: String
    titleEN: String
    totalIncome: Float
    typology: Typology
    url: String
    proposalId: String
    teams: [Team]!
    projectInstitutions: [ProjectInstitution]
    tec4Project: [TEC4Project]
    cae: String
    userCG: String
    createUser: String
    modifyUser: String
    createDate: String
    modifyDate: String
}

type Country {
    countryId: ID!
    countryAcronym: String
    countryCode: String
    countryDesignation: String
    countryDesignationEN: String
    createUser: String
    createDate: String
    modifyUser: String
    modifyDate: String
}

type FinancingEntity {
    entityId: ID!
    entityAcronym: String
    entityName: String
    entityNameEN: String
    country: Country
    valBegin: String
    valEnd: String
    createUser: String
    createDate: String
    modifyUser: String
    modifyDate: String
}

type Institution {
    institutionId: ID!
    contactEmail: String
    contactName: String
    contactTelephone: String
```

```
    institutionType: InstitutionType
    name: String
    shortName: String
    url: String
    nif: String
    cae: String
    address: String
    country: String
    city: String
    zipcode: String
    organizationalGroup: String
}

type InstitutionType {
    institutionTypeId: ID!
    name: String
}

type ProjectFunction {
    projectFunctionId: ID!
    name: String
}

type ProjectInstitution {
    institution: Institution
    project: Project
    approvedBudget: Float
    approvedCoFinancing: Float
    approvedFinancing: Float
    financingRate: Float
    confidential: Boolean
    projectInstitutionFunction: ProjectInstitutionFunction
    createUser: String
    modifyUser: String
    createDate: String
    modifyDate: String
}

type ProjectInstitutionFunction {
    projectInstitutionFunctionId: ID!
    name: String
}

type ProjectParticipation {
    projectParticipationId: ID!
    beginDate: String
    contributionProposal: Float
    contributionExecution: Float
    dedication: Float
    endDate: String
    highlight: Boolean
    idRH: Float
    professionalName: String
    projectFunction: ProjectFunction
    project: Project
    validated: Boolean
    eligible: Boolean
    affectation: Float
    createUser: String
    modifyUser: String
    createDate: String
    modifyDate: String
}

type Source {
    sourceId: ID!
    sourceAcronym: String
    sourceName: String
    sourceNameEN: String
    valBegin: String
    valEnd: String
```

```
    createUser: String
    createDate: String
    modifyUser: String
    modifyDate: String
}

type Structure {
    idEstrutura: ID!
    idTipoEstrutura: Float
    sigla: String
    nome: String
    nomeEN: String
    descricao: String
    descricaoEN: String
    telefone: String
    email: String
    cor: String
    oi: String
}

type Team {
    project: Project
    structure: Structure
    coordinator: Float
    oi: String
    coordinate: Boolean
    shortName: String
    totalBudget: Float
    cdBudget: Float
    cbBudget: Float
    createUser: String
    modifyUser: String
    createDate: String
    modifyDate: String
}

type TEC4 {
    tec4Id: Float
    beginDate: String
    endDate: String
    champion: String
    shortName: String
    description: String
    descriptionEN: String
    observations: String
    responsible: String
}

type TEC4 {
    tec4Id: Float
    beginDate: String
    endDate: String
    champion: String
    shortName: String
    description: String
    descriptionEN: String
    observations: String
    responsible: String
}

type Typology {
    typologyId: ID!
    typologyDescription: String
    typologyDescriptionEN: String
    observations: String
    typologyClass: TypologyClass
    financingEntity: FinancingEntity
    timecards: Boolean
    complements: Boolean
    valBegin: String
    valEnd: String
```

```
    createUser: String
    createDate: String
    modifyUser: String
    modifyDate: String
}

type TypologyClass {
    typologyClassId: ID!
    acronym: String
    designation: String
    designationEN: String
    source: Source
    valBegin: String
    valEnd: String
    createUser: String
    createDate: String
    modifyUser: String
    modifyDate: String
}


# The Root Query for the application
type Query {
    projectsAll: [Project]!
    projectById(projectId: ID!): Project
    projectByOIBase(OIBase: String!): Project
    financingEntityById(financingEntityId: ID!): FinancingEntity
    teamByProjectId(projectId: ID!): [Team]
    typologyById(typologyId: ID!): Typology
    typologyClassById(typologyClassId: ID!): TypologyClass

}
```

Appendix B – Prototype 1 POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pt.inesctec</groupId>
    <artifactId>prototype1</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <mssqlJDBC.version>6.1.0.jre8</mssqlJDBC.version>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphiql-spring-boot-starter</artifactId>
            <version>5.0.2</version>
        </dependency>

        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphql-spring-boot-starter</artifactId>
            <version>5.0.2</version>
        </dependency>
        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphql-java-tools</artifactId>
            <version>5.2.4</version>
        </dependency>
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.9</version>
        </dependency>

        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
```

```
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.microsoft.sqlserver</groupId>
            <artifactId>mssql-jdbc</artifactId>
            <version>7.0.0.jre8</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.4</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>0.8.2</version>
                <configuration>
                    <excludes>
                        <exclude>**/*pt/inesctec/prototype1/App.class</exclude>
                        <exclude>**/*pt/inesctec/prototype1/App**</exclude>
                        <exclude>**/*pt/inesctec/prototype1/data/entity/**</exclude>
                        <exclude>**/*pt/inesctec/prototype1/config/**</exclude>
<exclude>**/*pt/inesctec/prototype1/service/exceptionHandler/GraphQLErrorAdapter.class</e
xclude>
                    </excludes>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-agent</goal>
                        </goals>
                    </execution>
                    <!-- attached to Maven test phase -->
                    <execution>
                        <id>report</id>
                        <phase>test</phase>
                        <goals>
                            <goal>report</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>com.lazerycode.jmeter</groupId>
                <artifactId>jmeter-maven-plugin</artifactId>
                <version>2.7.0</version>
                <executions>
                    <execution>
                        <id>jmeter-tests</id>
                        <goals>
                            <goal>jmeter</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <jmeterExtensions>
                        <artifact>kg.apc:jmeter-plugins-casutg:2.4</artifact>

                        <artifactId>kg.apc:jmeter-plugins-extras-libs:1.3.1</artifactId>

                    </jmeterExtensions>
```

```
<testFilesDirectory>${project.basedir}/src/test/jmeter</testFilesDirectory>

<resultsDirectory>${project.basedir}/src/test/jmeter</resultsDirectory>
                    <downloadExtensionDependencies>false</downloadExtensionDependencies>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Appendix C – Prototype 2 POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pt.inesctec</groupId>
    <artifactId>prototype2</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <mssqlJDBC.version>6.1.0.jre8</mssqlJDBC.version>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphiql-spring-boot-starter</artifactId>
            <version>5.0.2</version>
        </dependency>

        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphql-spring-boot-starter</artifactId>
            <version>5.0.2</version>
        </dependency>
        <dependency>
            <groupId>com.graphql-java</groupId>
            <artifactId>graphql-java-tools</artifactId>
            <version>5.2.4</version>
        </dependency>
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.9</version>
        </dependency>

        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
```

```
            <artifactId>lombok</artifactId>
            <version>1.18.4</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>0.8.2</version>
                <configuration>
                    <excludes>
                        <exclude>**/*pt/inesctec/prototype2/App.class</exclude>
                        <exclude>**/*pt/inesctec/prototype2/App**</exclude>
                        <exclude>**/*pt/inesctec/prototype2/data/entity/**</exclude>
                        <exclude>**/*pt/inesctec/prototype2/config/**</exclude>
<exclude>**/*pt/inesctec/prototype2/service/exceptionHandler/GraphQLErrorAdapter.class</e
xclude>
                    </excludes>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-agent</goal>
                        </goals>
                    </execution>
                    <!-- attached to Maven test phase -->
                    <execution>
                        <id>report</id>
                        <phase>test</phase>
                        <goals>
                            <goal>report</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>com.lazerycode.jmeter</groupId>
                <artifactId>jmeter-maven-plugin</artifactId>
                <version>2.7.0</version>
                <executions>
                    <execution>
                        <id>jmeter-tests</id>
                        <goals>
                            <goal>jmeter</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <jmeterExtensions>
                        <artifact>kg.apc:jmeter-plugins-casutg:2.4</artifact>
                        <artifactId>kg.apc:jmeter-plugins-extras-libs:1.3.1</artifactId>
                    </jmeterExtensions>

<testFilesDirectory>${project.basedir}/src/test/jmeter</testFilesDirectory>

<resultsDirectory>${project.basedir}/src/test/jmeter</resultsDirectory>
                    <downloadExtensionDependencies>false</downloadExtensionDependencies>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Appendix D – Acceptance tests

Data	
Test Number	1
Test Goal	Validate the result from searching for a project using OI.
Input	Search OI “PG08011” and expected project id, title, and short name as a result.
Expected output	A JSON object with project id, title, and short name.
Real Results	
Prototype 1	A JSON object with project id, title, and short name.
Prototype 2	A JSON object with project id, title, and short name.

Data	
Test Number	2
Test Goal	Validate the result from searching for a project using OI that does not exist.
Input	Search OI “PG08000” and expected project id, title, and short name as a result.
Expected output	JSON object without data and with a project not found exception.
Real Results	
Prototype 1	JSON object without data and with a project not found exception.
Prototype 2	JSON object without data and with a project not found exception.

Data	
Test Number	3
Test Goal	Validate the result from searching for a project using OI with the wrong format.
Input	Search OI 8000 and expected project id, title, and short name as a result.
Expected output	JSON object without data and with an exception.
Real Results	
Prototype 1	JSON object without data and with an exception.
Prototype 2	JSON object without data and with an exception.

Data	
Test Number	4
Test Goal	Validate the result from searching for a project using project ID.
Input	Search ID 1 and expected project id, title, and short name as a result.
Expected output	A JSON object with project id, title, and short name.
Real Results	
Prototype 1	A JSON object with project id, title, and short name.
Prototype 2	A JSON object with project id, title, and short name.

Data	
Test Number	5
Test Goal	Validate the result from searching for a project using a project ID that does not exist.
Input	Search ID 500000 and expected project id, title, and short name as a result.
Expected output	JSON object without data and with a project not found exception.
Real Results	
Prototype 1	JSON object without data and with a project not found exception.
Prototype 2	JSON object without data and with a project not found exception.

Data	
Test Number	6
Test Goal	Validate the result from searching for a project using project ID with the wrong format.
Input	Search ID “xpto” and expected project id, title, and short name as a result.
Expected output	JSON object without data and with an exception.
Real Results	
Prototype 1	JSON object without data and with an exception.
Prototype 2	JSON object without data and with an exception.

Data	
Test Number	7
Test Goal	Validate the result from searching for all the projects.
Input	Search all and expected project id, title, and short name as a result.
Expected output	A JSON object with project id, title, and short name.
Real Results	
Prototype 1	A JSON object with project id, title, and short name.
Prototype 2	A JSON object with project id, title, and short name.

Data	
Test Number	8
Test Goal	Validate the result from searching a financing entity using entity ID.
Input	Search ID 1 and expected entity id and entity acronym as a result.
Expected output	A JSON object with project id, title, and short name.
Real Results	
Prototype 1	A JSON object with entity id and entity acronym.
Prototype 2	A JSON object with entity id and entity acronym.

Data	
Test Number	9
Test Goal	Validate the result from searching a financing entity using entitiy ID that does not exist.
Input	Search ID 999999 and expected entity id and entity acronym as a result.
Expected output	JSON object without data and with a financing entity not found exception.
Real Results	
Prototype 1	JSON object without data and with a financing entity not found exception.
Prototype 2	JSON object without data and with a financing entity not found exception.

Data	
Test Number	10
Test Goal	Validate the result from searching a financing entity using entity ID with the wrong format.
Input	Search ID “xpto” and expected entity id and entity acronym as a result.
Expected output	JSON object without data and with an exception.
Real Results	
Prototype 1	JSON object without data and with an exception.
Prototype 2	JSON object without data and with an exception.

Data	
Test Number	11
Test Goal	Validate the result from searching for a team using project ID.
Input	Search ID 1 and expected project id, structure acronym, coordinator, and short name as a result.
Expected output	A JSON object with project id, structure acronym, coordinator, and short name.
Real Results	
Prototype 1	A JSON object with project id, structure acronym, coordinator, and short name.
Prototype 2	A JSON object with project id, structure acronym, coordinator, and short name.

Data	
Test Number	12
Test Goal	Validate the result from searching for a team using project ID that does not exist.
Input	Search ID 999999 and expected project id, structure acronym, coordinator, and short name as a result.
Expected output	JSON object without data and with a team not found exception.
Real Results	
Prototype 1	JSON object without data and with a team not found exception.
Prototype 2	JSON object without data and with a team not found exception.

Data	
Test Number	13
Test Goal	Validate the result from searching a team using project ID with the wrong format.
Input	Search ID “xpto” and expected project id, structure acronym, coordinator, and short name as a result.
Expected output	JSON object without data and with an exception.
Real Results	
Prototype 1	JSON object without data and with an exception.
Prototype 2	JSON object without data and with an exception.

Data	
Test Number	14
Test Goal	Validate the result from searching a typology using typology ID.
Input	Search ID 1 and expected typology id, typology class id, and typology description as a result.
Expected output	A JSON object with typology id, typology class id, and typology description.
Real Results	
Prototype 1	A JSON object with typology id, typology class id, and typology description.
Prototype 2	A JSON object with typology id, typology class id, and typology description.

Data	
Test Number	15
Test Goal	Validate the result from searching a typology using typology ID that does not exist.
Input	Search ID 999999 and expected typology id, typology class id, and typology description as a result.
Expected output	JSON object without data and with a typology not found exception.
Real Results	
Prototype 1	JSON object without data and with a typology not found exception.
Prototype 2	JSON object without data and with a typology not found exception.

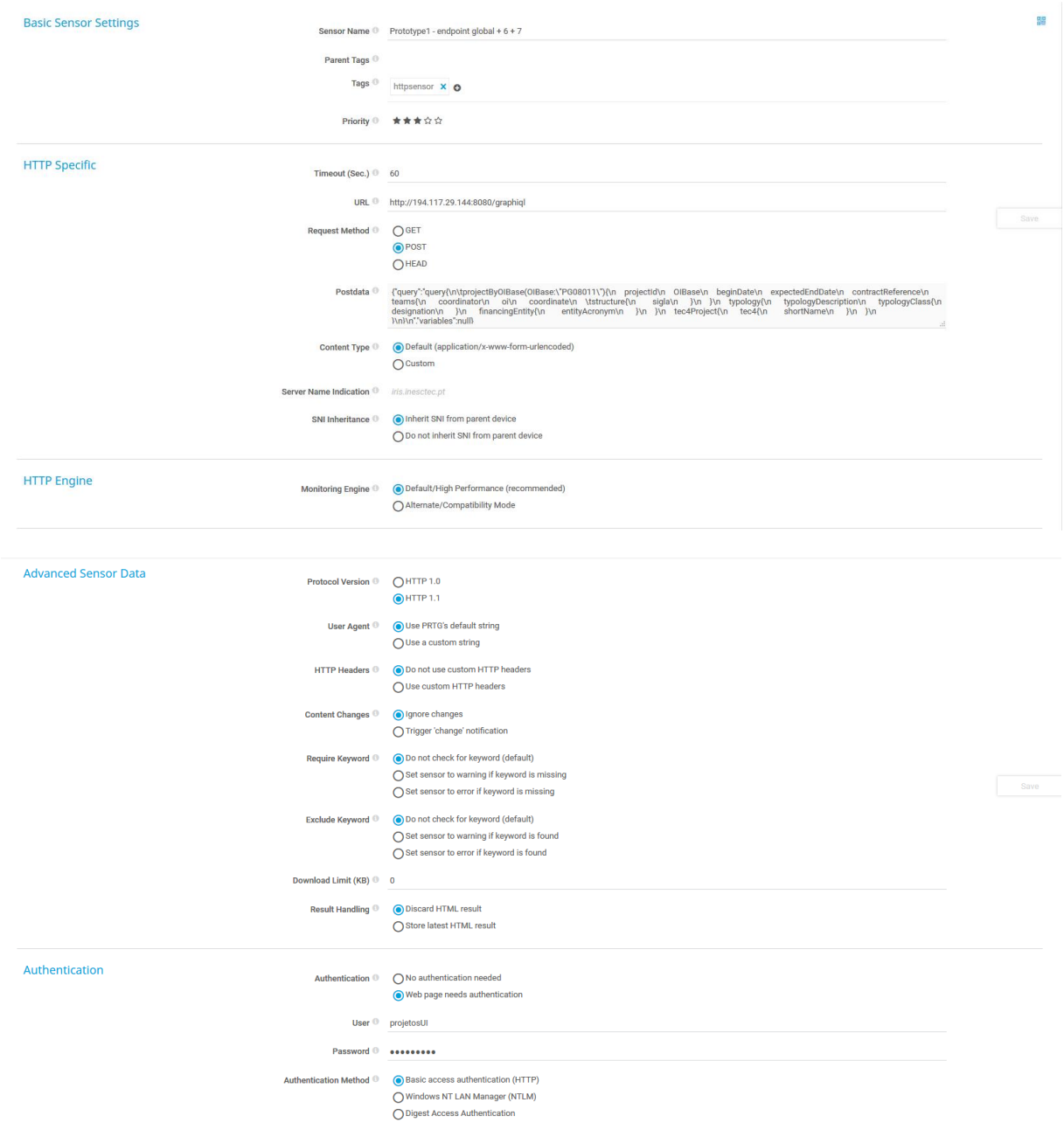
Data	
Test Number	16
Test Goal	Validate the result from searching a typology using typology ID with the wrong format.
Input	Search ID “xpto” and expected typology id, typology class id, and typology description as a result.
Expected output	JSON object without data and with an exception.
Real Results	
Prototype 1	JSON object without data and with an exception.
Prototype 2	JSON object without data and with an exception.

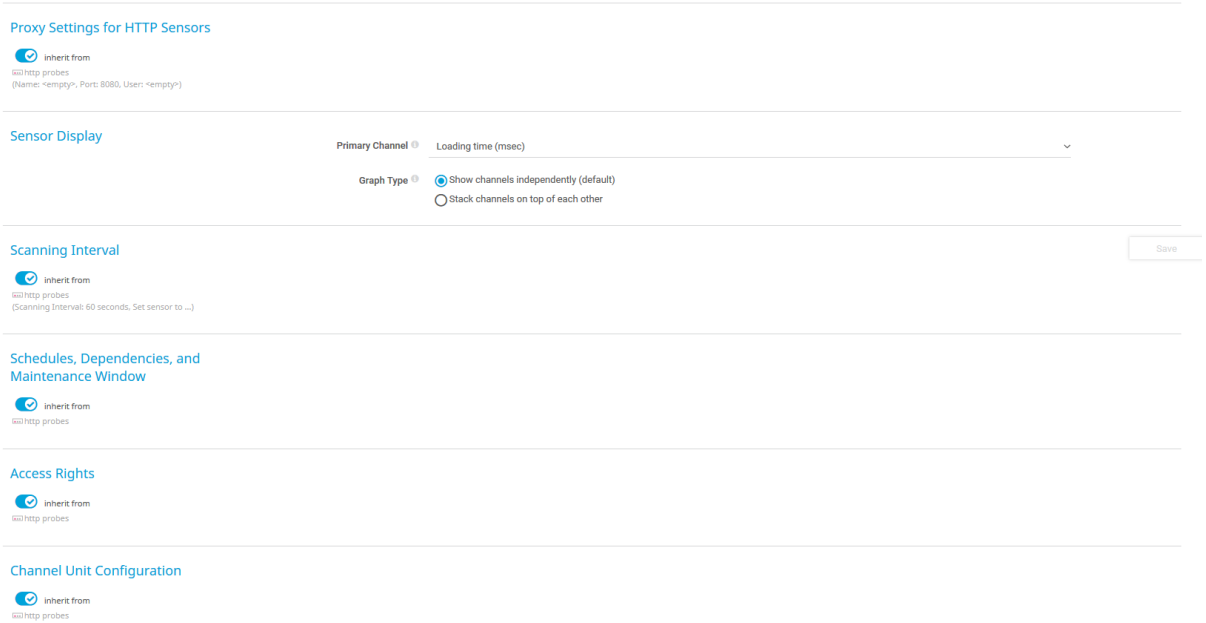
Data	
Test Number	17
Test Goal	Validate the result from searching a typology class using typology class ID.
Input	Search ID 1 and expected typology class id and typology description as a result.
Expected output	A JSON object with typology class id and acronym.
Real Results	
Prototype 1	A JSON object with typology class id and acronym.
Prototype 2	A JSON object with typology class id and acronym.

Data	
Test Number	18
Test Goal	Validate the result from searching a typology class using typology class ID that does not exist.
Input	Search ID 999999 and expected typology class id and acronym as a result.
Expected output	JSON object without data and with a typology class not found exception.
Real Results	
Prototype 1	JSON object without data and with a typology class not found exception.
Prototype 2	JSON object without data and with a typology class not found exception.

Data	
Test Number	19
Test Goal	Validate the result from searching a typology class using typology class ID with the wrong format.
Input	Search ID “xpto” and expected typology class id and acronym as a result.
Expected output	JSON object without data and with an exception.
Real Results	
Prototype 1	JSON object without data and with an exception.
Prototype 2	JSON object without data and with an exception.

Appendix E – Example of a sensor configuration




Proxy Settings for HTTP Sensors
inherit from
http probes
(Name: <empty>, Port: 8080, User: <empty>)
Sensor Display
Primary Channel
Loading time (msec)
Graph Type
Show channels independently (default)
Stack channels on top of each other
Scanning Interval
Save
inherit from
http probes
(Scanning Interval: 60 seconds, Set sensor to ...)
Schedules, Dependencies, and Maintenance Window
inherit from
http probes
Access Rights
inherit from
http probes
Channel Unit Configuration
inherit from
http probes

Appendix F – Descriptive statistics (quantitative data) for time

Summary statistics (Quantitative data):

Variable	Observations	Obs. with missing data	Obs. without missing data	Minimum	Maximum	Mean	Std. deviation
Y	27	0	27	128	13480	2279	4209

Summary statistics (Qualitative data):

Variable	Categories	Counts	Frequencies	%
API	Current Solution	9	9	33
	Prototype 1	9	9	33
	Prototype 2	9	9	33

Correlation matrix:

	API-Current Solutic	API-Prototype 1	API-Prototype 2	Y
API-Current Solut	**1**	-1	-1	1
API-Prototype 1	-1	**1**	-1	0
API-Prototype 2	-1	-1	**1**	0
Y	1	0	0	**1**

Regression of variable Y:

Goodness of fit statistics (Y):

Observations	27
Sum of weights	27
DF	24
R^2	0
Adjusted R^2	0
MSE	11306234
RMSE	3362
MAPE	553
DW	2
Cp	3
AIC	441
SBC	445
PC	1

Analysis of variance (Y):

Source	DF	Sum of squares	Mean squares	F	Pr > F
Model	2	189331420	94665710	8	**0**
Error	24	271349618	11306234		
Total corrected	26	460681039			

Computed against model Y=Mean(Y)

Type I Sum of Squares analysis (Y):

Source	DF	Sum of squares	Mean squares	F	Pr > F
API	2	189331420	94665710	8	**0**

Type II Sum of Squares analysis (Y):

Source	DF	Sum of squares	Mean squares	F	Pr > F
API	2	189331420	94665710	8	**0**

Type III Sum of Squares analysis (Y):

Source	GL	Sum of squares	Mean squares	F	Pr > F
API	2	189331420	94665710	8	**0**

Model parameters (Y):

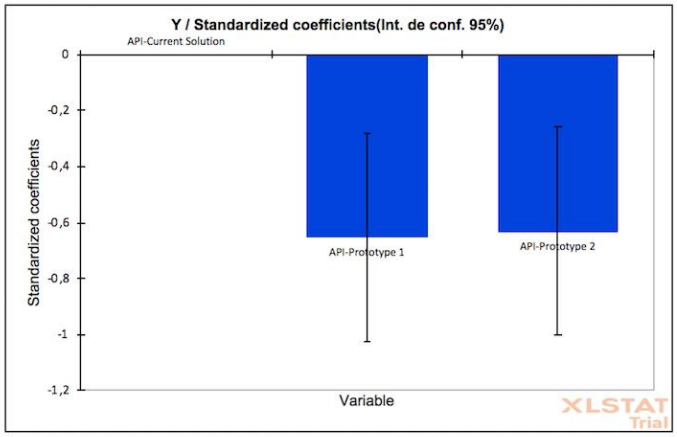
Source	Value	Standard error	t	Pr > |t|	Lower bound (95%)	Upper bound (95%)
Intercept	6022	1121	5	**< 0,0001**	3709	8335
API-Current Solution	0	0				
API-Prototype 1	-5710	1585	-4	**0**	-8981	-2438
API-Prototype 2	-5520	1585	-3	**0**	-8792	-2249

Equation of the model (Y):

Y = 6022,11111111111-5709,88888888889*API-Prototype 1-5520,11111111111*API-Prototype 2

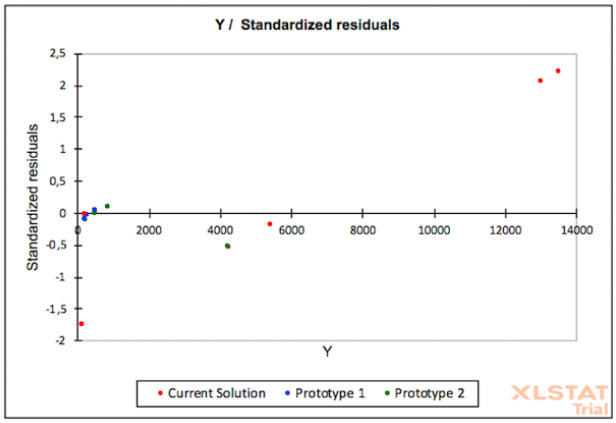
Standardized coefficients (Y):

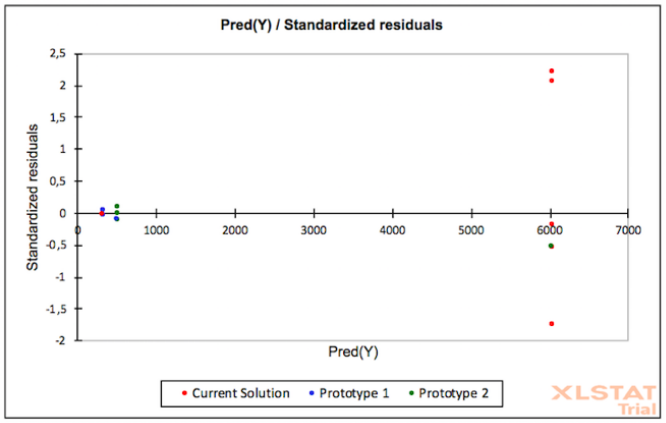
Source	Value	Standard error	t	Pr > |t|	Lower bound (95%)	Upper bound (95%)
API-Current Solution	0	0				
API-Prototype 1	-1	0	-4	**0**	-1	0
API-Prototype 2	-1	0	-3	**0**	-1	0

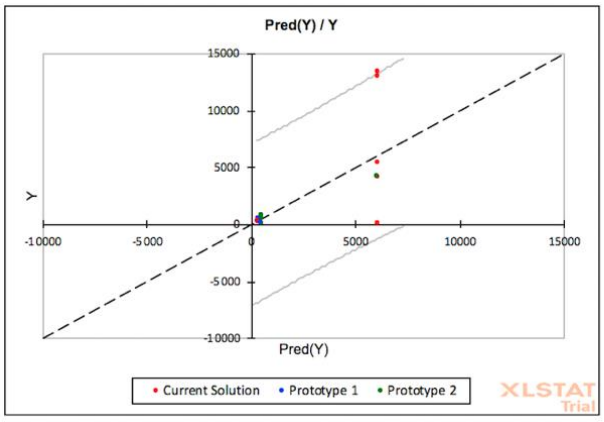


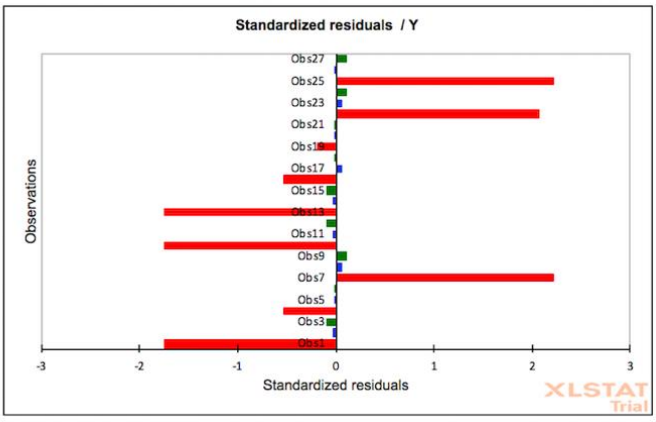
Predictions and residuals (Y):

Observation	Weight	Y	Pred(Y)	Residual	Std. residual	Std. dev. on pred. (Mean)	Lower bound 95% (Mean)	Upper bound 95% (Mean)	Std. dev. on pred. (Observation)	Lower bound 95% (Observation)	Upper bound 95% (Observation)
Obs1	1	129	6022	-5893	-2	1121	3709	8335	3544	-1293	13337
Obs2	1	203	312	-109	0	1121	-2001	2625	3544	-7003	7627
Obs3	1	195	502	-307	0	1121	-1811	2815	3544	-6813	7817
Obs4	1	4236	6022	-1786	-1	1121	3709	8335	3544	-1293	13337
Obs5	1	255	312	-57	0	1121	-2001	2625	3544	-7003	7627
Obs6	1	480	502	-22	0	1121	-1811	2815	3544	-6813	7817
Obs7	1	13480	6022	7458	2	1121	3709	8335	3544	-1293	13337
Obs8	1	480	312	168	0	1121	-2001	2625	3544	-7003	7627
Obs9	1	830	502	328	0	1121	-1811	2815	3544	-6813	7817
Obs10	1	128	6022	-5894	-2	1121	3709	8335	3544	-1293	13337
Obs11	1	202	312	-110	0	1121	-2001	2625	3544	-7003	7627
Obs12	1	196	502	-306	0	1121	-1811	2815	3544	-6813	7817
Obs13	1	129	6022	-5893	-2	1121	3709	8335	3544	-1293	13337
Obs14	1	203	312	-109	0	1121	-2001	2625	3544	-7003	7627
Obs15	1	195	502	-307	0	1121	-1811	2815	3544	-6813	7817
Obs16	1	4237	6022	-1785	-1	1121	3709	8335	3544	-1293	13337
Obs17	1	480	312	168	0	1121	-2001	2625	3544	-7003	7627
Obs18	1	483	502	-19	0	1121	-1811	2815	3544	-6813	7817
Obs19	1	5400	6022	-622	0	1121	3709	8335	3544	-1293	13337
Obs20	1	254	312	-58	0	1121	-2001	2625	3544	-7003	7627
Obs21	1	480	502	-22	0	1121	-1811	2815	3544	-6813	7817
Obs22	1	12980	6022	6958	2	1121	3709	8335	3544	-1293	13337
Obs23	1	478	312	166	0	1121	-2001	2625	3544	-7003	7627
Obs24	1	830	502	328	0	1121	-1811	2815	3544	-6813	7817
Obs25	1	13480	6022	7458	2	1121	3709	8335	3544	-1293	13337
Obs26	1	255	312	-57	0	1121	-2001	2625	3544	-7003	7627
Obs27	1	829	502	327	0	1121	-1811	2815	3544	-6813	7817







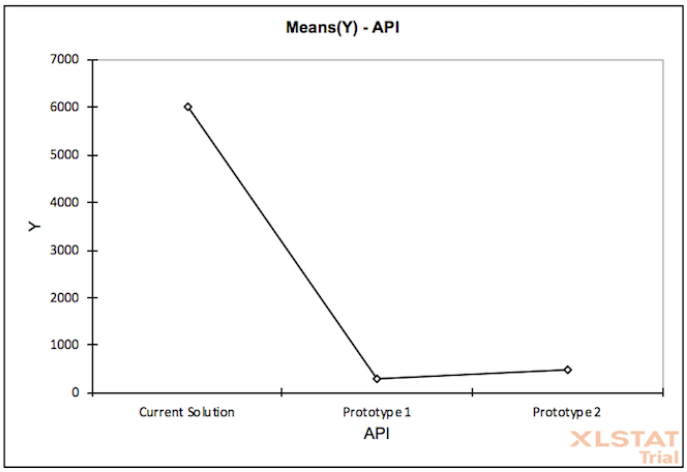


Interpretation (Y):

Given R2, 41% of the variability of the dependent variable Y was explained by the explanatory variable.

Given the p-value of the F-statistic calculated in the ANOVA table, and given the significance level of 5%, the information brought by the explanatory variables was significantly better than a baseline average.

Means graphic:



API / Tukey (HSD) / Analysis of the differences between the categories with a confidence interval of 95% (Y):

Constrast	Difference	Standardized difference	Critical value	Pr > Diff	Significant	Inferior limit(95%)	Superior limit (95%)	Inferior limit(95%)	Superior limit (95%)
Current Solution vs Prototype 1	5710	4	2	**0**	Yes	1751	9668		
Current Solution vs Prototype 2	5520	3	2	**0**	Yes	1562	9479		
Prototype 2 vs Prototype 1	190	0	2	1	No	-3769	4148		
Tukey's d critical value:			4						

Category	LS means	Standard error	Groups	
Current Solution	6022	1121	A	
Prototype 2	502	1121		B
Prototype 1	312	1121		B

API / Dunnett (bilateral) / Analysis of the differences between the control categorie API-Current Solution and the other categories with a confidence interval of 95%:

Constrast	Difference	Standardized difference	Critical value	tical differer	Pr > Diff	Significant
Current Solution vs Prototype 1	5710	4	2	3724	0	Yes
Current Solution vs Prototype 2	5520	3	2	3724	0	Yes

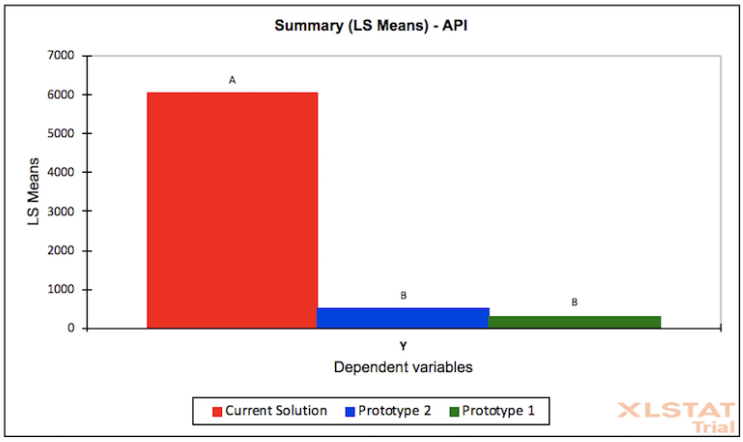
Summary of paired comparisons for API (Tukey (HSD)):

Category	LS means (Y)	Groups	
Current Solution	6022	A	
Prototype 2	502		B
Prototype 1	312		B

Summary (LS means) - API:

	Y
Current Solution	6022, a
Prototype 2	502, b
Prototype 1	312, b
Pr > F(Model)	0
Significant	Yes
Pr > F(API)	0
Significant	Yes

	Y
Current Solution	6022
Prototype 2	502
Prototype 1	312



Appendix G – Descriptive statistics (quantitative data) for size

Summary statistics (Quantitative data):

Variable	Observations	Obs. with missing data	Obs. without missing data	Minimum	Maximum	Mean	Std. deviation
Y	27	0	27	0	29	5	9

Summary statistics (Qualitative data):

Variable	Categories	Counts	Frequencies	%
API	Current Solution	9	9	33
	Prototype 1	9	9	33
	Prototype 2	9	9	33

Correlation matrix:

	API-Current Solution	API-Prototype 1	API-Prototype 2	Y
API-Current Solution	**1**	-1	-1	1
API-Prototype 1	-1	**1**	-1	0
API-Prototype 2	-1	-1	**1**	0
Y	1	0	0	**1**

Regression of variable Y:

Goodness of fit statistics (Y):

Observations	27
Sum of weights	27
DF	24
R^2	0
Adjusted R^2	0
MSE	56
RMSE	7
MAPE	401
DW	2
Cp	3
AIC	111
SBC	115
PC	1

Analysis of variance (Y):

Source	DF	Sum of squares	Mean squares	F	Pr > F
Model	2	767	383	7	**0**
Error	24	1343	56		
Total corrected	26	2110			

Computed against model Y=Mean(Y)

Type I Sum of Squares analysis (Y):

Source	DF	Sum of squares	Mean squares	F	Pr > F
API	2	767	383	7	**0**

Type II Sum of Squares analysis (Y):

Source	DF	Sum of squares	Mean squares	F	Pr > F
API	2	767	383	7	**0**

Type III Sum of Squares analysis (Y):

Source	GL	Sum of squares	Mean squares	F	Pr > F
API	2	767	383	7	**0**

Model parameters (Y):

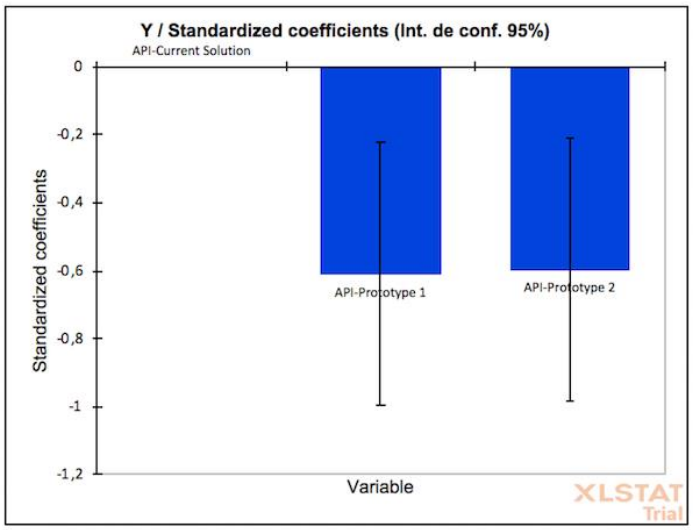
Source	Value	Standard error	t	Pr > |t|	Lower bound (95%)	Upper bound (95%)
Intercept	12	2	5	**< 0,0001**	7	17
API-Current Solution	0	0				
API-Prototype 1	-11	4	-3	**0**	-19	-4
API-Prototype 2	-11	4	-3	**0**	-18	-4

Equation of the model (Y):

Y = 12,0722222222222-11,4422222222222*API-Prototype 1-11,1622222222222*API-Prototype 2

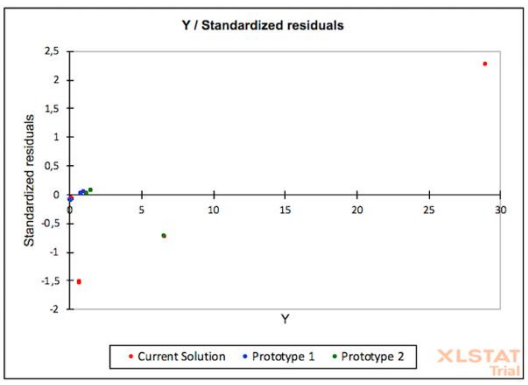
Standardized coefficients (Y):

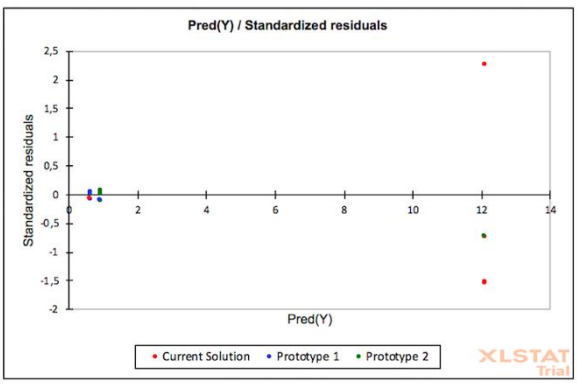
Source	Value	Standard error	t	Pr > |t|	Lower bound (95%)	Upper bound (95%)
API-Current Solution	0	0				
API-Prototype 1	-1	0	-3	**0**	-1	0
API-Prototype 2	-1	0	-3	**0**	-1	0

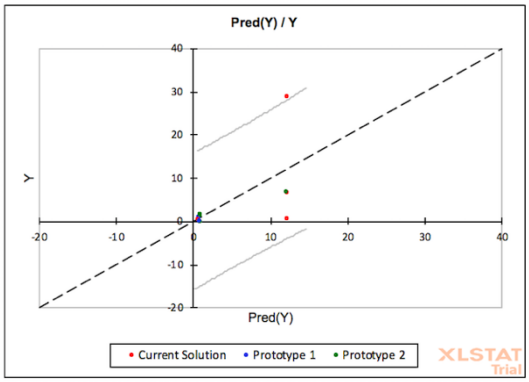


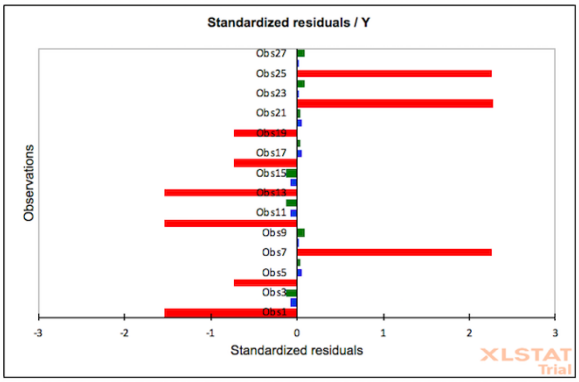
Predictions and residuals (Y):

Observation	Weight	Y	Pred(Y)	Residual	Std. residual	Std. dev. on pred. (Mean)	Lower bound 95% (Mean)	Upper bound 95% (Mean)	Std. dev. on pred. (Observation)	Lower bound 95% (Observation)	Upper bound 95% (Observation)
Obs1	1	1	12	-11	-2	2	7	17	8	-4	28
Obs2	1	0	1	0	0	2	-5	6	8	-16	17
Obs3	1	0	1	-1	0	2	-4	6	8	-15	17
Obs4	1	7	12	-5	-1	2	7	17	8	-4	28
Obs5	1	1	1	0	0	2	-5	6	8	-16	17
Obs6	1	1	1	0	0	2	-4	6	8	-15	17
Obs7	1	29	12	17	2	2	7	17	8	-4	28
Obs8	1	1	1	0	0	2	-5	6	8	-16	17
Obs9	1	2	1	1	0	2	-4	6	8	-15	17
Obs10	1	1	12	-11	-2	2	7	17	8	-4	28
Obs11	1	0	1	0	0	2	-5	6	8	-16	17
Obs12	1	0	1	-1	0	2	-4	6	8	-15	17
Obs13	1	1	12	-11	-2	2	7	17	8	-4	28
Obs14	1	0	1	0	0	2	-5	6	8	-16	17
Obs15	1	0	1	-1	0	2	-4	6	8	-15	17
Obs16	1	7	12	-5	-1	2	7	17	8	-4	28
Obs17	1	1	1	0	0	2	-5	6	8	-16	17
Obs18	1	1	1	0	0	2	-4	6	8	-15	17
Obs19	1	7	12	-5	-1	2	7	17	8	-4	28
Obs20	1	1	1	0	0	2	-5	6	8	-16	17
Obs21	1	1	1	0	0	2	-4	6	8	-15	17
Obs22	1	29	12	17	2	2	7	17	8	-4	28
Obs23	1	1	1	0	0	2	-5	6	8	-16	17
Obs24	1	2	1	1	0	2	-4	6	8	-15	17
Obs25	1	29	12	17	2	2	7	17	8	-4	28
Obs26	1	1	1	0	0	2	-5	6	8	-16	17
Obs27	1	2	1	1	0	2	-4	6	8	-15	17







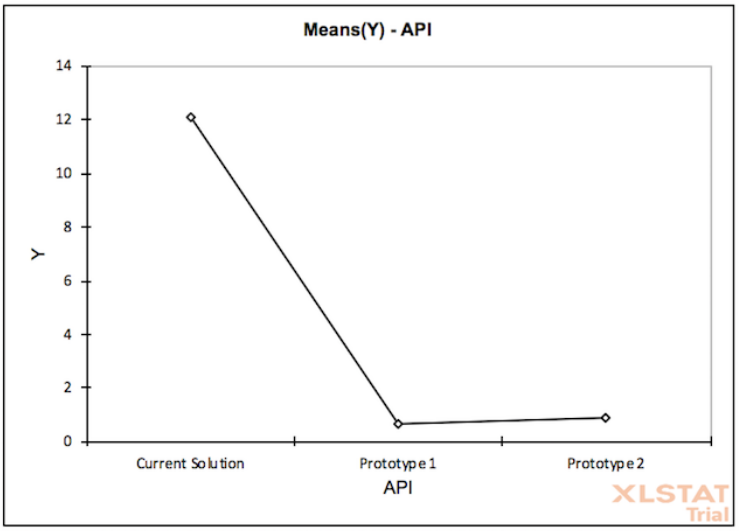


Interpretation (Y):

Given the R2, 36% of the variability of the dependent variable Y was explained by the explanatory variable.

Given the p-value of the F-statistic calculated in the ANOVA table, and given the significance level of 5%, the information brought by the explanatory variables was significantly better than a baseline average.

Means graphic:



API / Tukey (HSD) / Analysis of the differences between the categories with a confidence interval of 95% (Y):

Constrast	Difference	Standardized difference	Critical value	Pr > Diff	Significant	Inferior limit(95%)	Superior limit (95%)	Inferior limit(95%)	Superior limit (95%)
Current Solution vs Prototype 1	11	3	2	**0**	Yes	3	20		
Current Solution vs Prototype 2	11	3	2	**0**	Yes	2	20		
Prototype 2 vs Prototype 1	0	0	2	1	No	-9	9		
Tukey's d critical value:			4						

Category	LS means	Standard error	Groups
Current Solution	12	2	A
Prototype 2	1	2	B
Prototype 1	1	2	B

API / Dunnett (bilateral) / Analysis of the differences between the control categorie API-Current Solution and the other categories with a confidence interval of 95%:

Constrast	Difference	Standardized difference	Critical value	Critical difference	Pr > Diff	Significant
Current Solution vs Prototype 1	11	3	2	8	0	Yes
Current Solution vs Prototype 2	11	3	2	8	0	Yes

Summary of paired comparisons for API (Tukey (HSD)):

Category	LS means (Y)	Groups	
Current Solution	12	A	
Prototype 2	1		B
Prototype 1	1		B

Summary (LS means) - API:

	Y
Current Solution	12, a
Prototype 2	1, b
Prototype 1	1, b
Pr > F(Model)	0
Significant	Yes
Pr > F(API)	0
Significant	Yes

	Y
Current Solution	12
Prototype 2	1
Prototype 1	1