Vázquez-Ingelmo, A., Cruz-Benito, J., and García-Peñalvo, F.J., 2017. Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL. In *Fifth International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM'17) (Cádiz, Spain, October 18-20, 2017)* J.M. Dodero, M.S. Ibarra Sáiz and I. Ruiz Rube Eds. ACM, New York, NY, USA, Article 89. DOI= http://dx.doi.org/10.1145/3144826.3145437

Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL

Andrea Vázquez-Ingelmo
GRIAL Research Group, Department of Computer and Automatics, University of Salamanca
Paseo de Canalejas 169, 37008 Salamanca, Spain
andreavazquez@usal.es

Juan Cruz-Benito
GRIAL Research Group, Department of Computer and Automatics, University of Salamanca
Paseo de Canalejas 169, 37008 Salamanca, Spain
juancb@usal.es

Francisco J. García-Peñalvo
GRIAL Research Group, Department of Computer and Automatics, University of Salamanca
Paseo de Canalejas 169, 37008 Salamanca, Spain
fgarcia@usal.es

ABSTRACT

A crucial part of data-driven ecosystems is the management and processing of complex data structures, as well as the proper handling of the data flows within the ecosystem. To manage these data flows, data-driven ecosystems need high levels of interoperability, as it allows the collaboration and independence of both internal and external components. REST APIs are a common solution to achieve interoperability, but sometimes they lack flexibility and performance. The arising of GraphQL APIs as a flexible, fast and stable protocol for data fetching makes it an interesting approach for data-intensive and complex data-driven (eco)systems. This paper outlines the GraphQL protocol and the benefits derived from its use, as well as it presents a case of study of the improvement experienced by the Observatory of Employment and Employability (also known as *OEEU*) ecosystem after including GraphQL as main API in several components. The results of the paper show promising improvements regarding the flexibility, maintainability and performance, among other benefits.

KEYWORDS

GraphQL; API; Technological ecosystems; Data-driven; Interoperability

1 INTRODUCTION

Data-driven [1, 2] technological ecosystems have to continually deal with complex data structures and different data flows in order to achieve the ecosystem's main purpose. These data-driven ecosystems need to rely on a collaborative technological environment made up of different components that gather, analyze and disseminate the problem's domain data [3].

Due to the existence of different and heterogeneous components with separated tasks within technological ecosystems, it is important having the support of communication methods and high levels of interoperability to reach the components' collaboration without losing their own independence [4].

The implementation of REST [5] APIs (Application Programming Interface) to retrieve, create or modify the ecosystem's data fosters high levels of interoperability by creating well-defined interfaces and endpoints for data transactions. REST APIs decouple data consumers from data sources, connecting them through data flows independently of their platforms or technical characteristics.

However, REST APIs - although well implemented - can present lack of flexibility. They must have endpoints for the data requests contemplated in the requirements. But requirements can evolve through time, as well as the domain data's structure and the ecosystem's components (or users), making it necessary to rewrite the REST API interfaces.

Even if the REST APIs' interfaces evolve along with the requirements, some components (or users) might need only certain parts or fields of the whole (eco)system's data, or even a combination of fields that belong to different data objects, and find out that none of the implemented REST API's endpoints satisfies their specific request. In that case, the components (or users) need to make a request to the endpoint that returns the closest set of fields required or make a series of requests to different endpoints to gather all the data they need [6].

The previous scenario could be solved by implementing an endpoint for every possible data request, but, again, this will result in flexibility and maintainability issues.

This flexibility, maintainability and performance problems derived from REST APIs are what the GraphQL language is trying to solve. GraphQL (https://facebook.github.io/graphql/) is "a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions" [7].

The main goal of GraphQL is to unify the data requests of an (eco)system into one unique endpoint, reducing the number of requests needed to retrieve data. On the other hand, GraphQL allows components and users to specify in their own requests the particular fields they are asking for, retrieving solely what they need for their purposes.

In data-driven technological ecosystems, the components can individually and collectively evolve through time, but they need to keep collaborate in order to achieve the whole ecosystem's purpose [4]. That is why the features of GraphQL makes this query language suitable for data-driven technological ecosystems, as it provides interoperability (necessary for the components' communication and their data flows), flexibility (making smoother the ecosystem's continuous evolution) and also it improves performance by reducing the number and size (most of the times in broadband) of data requests.

The rest of this paper is structured as follow: the second section outlines de basics of the GraphQL protocol and its queries; the section 3 presents the data issues related to the Observatory of Employment and Employability's ecosystem and describes the APIs evolution from a REST API to a GraphQL API; section 4 shows the results and benefits resulting from the implementation of the GraphQL API in the Observatory's ecosystem. Finally, the fifth section discusses the problems addressed by the change, followed by the sixth and last section, where the conclusions reached are presented.

2 GRAPHQL BASICS

As explained before, GraphQL is a query language located in the application layer, which means that it can be used against any backend that meet the protocol's specification [7].

In GraphQL, "products are described in graphs and queries, instead of the REST notion of endpoints" [8]. This "graph nature" is what makes (eco)systems that implement GraphQL more flexible and scalable. The modification of the data objects enclosed in this new API paradigm is similar to the modification of a graph; the addition or removal of data fields is trivial because the GraphQL approach consider these operations as additions or removals of nodes in a graph, simplifying the modifications and evolution of the domain's data.

With the GraphQL approach, data instances are represented as a set of fields (nodes), which also can have nested fields and relations with other nodes. It is the responsibility of the GraphQL backend to specify the data objects and the fields that are available for the clients to query.

Keeping in mind the graph nature of the language and the relations between nodes, a GraphQL query is a set of hierarchical fields of data that a client wants to retrieve, and as a result, only that specified fields will be returned.

GraphQL queries are performed against a unique endpoint which unifies all possible data requests. That means that only one URL is needed to serve all the (eco)system's data available through the API [7]. This approach allows the clients to design their specific queries, and frees the (eco)systems and its developers from taking into account and implement endpoints for all the possible data requests combinations, resulting in an increase of flexibility and maintainability.

A simple GraphQL query example is presented in the Fig. 1. Querying it against a server that supports GraphQL (and holds information about people) would result in a response with the structure presented in the Fig. 2. These queries are sent to the target backend through an HTTP POST request, where the payload is the JSON-structured query specified by the client.


```
{
    person(id: some_id)
    {
        name,
        age,
        height,
        weight,
        parents
        {
            name
        }
    }
}
```

Figure 1: Example of a GraphQL query executed to retrieve information of a specific person.

```
{
    "person"
    {
        "name": "John",
        "age": 23,
        "height": 1.75,
        "weight": 63,
        "parents": [
            {
                "name": "Alice"
            },
            {
                "name": "Bob"
            }
        ]
    }
}
```

Figure 2: Example of a possible GraphQL query response with information of a specific person.

The GraphQL queries are client-specific [7], which means that the response's structure is led by the initial query's structure, and the results are returned in the same order as requested, providing a lot of freedom to the client.

If the clients' data requirements change at some point and they need to retrieve other fields, they simply have to modify their query's structure and send it to the same endpoint.

Another characteristic of GraphQL is that it is strong-typed; the types of the returned response's fields are specified at the backend. By this way, consumers are able to know the shape and nature of the responses.

GraphQL also provides mechanisms to filter results. As pointed out in the GraphQL specification, the query's fields can be seen as "functions" (implemented at the backend) that returns values; and as "functions", they can accept arguments to alter their behavior or the returned values themselves [7]. For instance, in the Fig. 1, "id" is an argument that tells the backend which particular person the client is requesting. Of course, there must be mechanisms implemented in the backend to filter the data objects by an identifier and, consequently, make the argument "id" available.

But this language not only provides a flexible and efficient manner to read data from backend sources, it also allows the four basic functions of CRUD: create, read, update and delete. In contrast with REST, where the CRUD functions are specified in the HTTP method (PUT, POST, GET and DELETE), GraphQL defines three types of operations [7] independent of the HTTP request:

- Queries: read-only retrieval (already explained).
- Mutation: a modification on the backend followed by a response with results.
- Subscriptions: requests in response to source events.

An example of a mutation can be seen in the Fig. 3. Once this mutation request arrives to the backend, a new data object will be created in the database with the fields provided (name and age in this example), and the client would receive a response with the identifier of the person created (or with any other field specified in the initial request). The backend is responsible of implementing the operations available (create, update, delete, etc.).

```
mutation {
  createPerson(input: {
    name: "Alan",
    age: 32,
  }) {
    id
  }
}
```

Figure 3: Example of a GraphQL mutation used to create a new person named "Alan" in the backend.

Subscriptions, on the other hand, are data responses that are sent to the clients subscribed when a particular event fires.

These are the basic characteristics and concepts behind GraphQL for the scope of this paper, for a wider view of this query language and all its possibilities, authors refer to its technical specification [7].

3 CASE OF STUDY: THE OEEU'S DATA-DRIVEN ECOSYSTEM

A GraphQL API has been implemented for the OEEU's data-driven ecosystem to test the benefits derived from the use of this language.

The first subsection introduces the Observatory of Employability and Employment and its data handling issues.

Subsections 3.2 and 3.3 outline the Observatory's REST API (and its limitations) and the Observatory's GraphQL API implemented to improve the collaboration between their data-driven ecosystem's components, respectively.

3.1 The OEEU's Problem

The Observatory of University Employability and Employment (also known as *OEEU* using the Spanish initials for *Observatorio de Empleabilidad y Empleo Universitarios*) http://oeeu.org/, is an organization formed by a group of researchers and technicians with the vision of becoming the information reference for understanding and exploiting knowledge about variables related to employability and university employment (and its behavior) [9, 10].

The organization itself is data-driven, as all its activities are backed up by data. To achieve its goals, the Observatory needs to manage significant amounts of data from different information sources, including Spanish universities and their students. But raw data is not enough to reach insights about employment and employability; all the collected data must be analyzed to allow evidence-based decision making.

With the evidences and knowledge gained by the analysis of the gathered data, the Observatory's studies can help universities to improve its policies, metrics and results regarding their students' employability, as well as their preparation and satisfaction with those important aspects. This analysis of organizational and students' data from universities frames the Observatory in the scope of areas like Academic Analytics [11] and knowledge management in the university scope [12, 13].

The Observatory's studies are supported by a data-driven technological ecosystem formed by a series of components responsible for collecting, storing, analyzing, disseminating and visualizing the domain's data [2, 14]. This ecosystem is based on the collaboration of different components to achieve the purposes of the Observatory.

However, employment and employability data is continuously increasing, and the issues related to them regarding the technological ecosystem needs to stick to evolving requirements while maintaining a collaborative environment.

The transition from a monolithic system to a data-driven ecosystem through the improvement of the interoperability levels between its components (by the implementation of a REST API) made the Observatory's technological support a powerful tool. But, as introduced before, REST APIs could not provide enough flexibility, scalability and performance levels for such changing study fields and technologies.

3.2 The OEEU's REST API

One of the main goals of the Observatory's ecosystem is to disseminate the knowledge gained by the analysis of the gathered data. To accomplish this goal, the Observatory's system includes different presentation and data analysis components that provide support to reach insights about graduates' employability.

The purpose of the presentation components in the ecosystem is to show information through tables and visualizations after the raw data has been analyzed by the data analyzer component. It is obvious that it was necessary to connect these components to make the analyzed information available on demand for its visualization. But not only the communication between these components was a concern; other components of the Observatory's ecosystem also needed communication methods to handle their data flows inputs and outputs.

Furthermore, the Observatory's vision of becoming an information source reference for employment and university employability needed communication methods to allow the connection of external components and to create a wider technological environment to gain more knowledge and wisdom through the collaboration of different information (eco)systems.

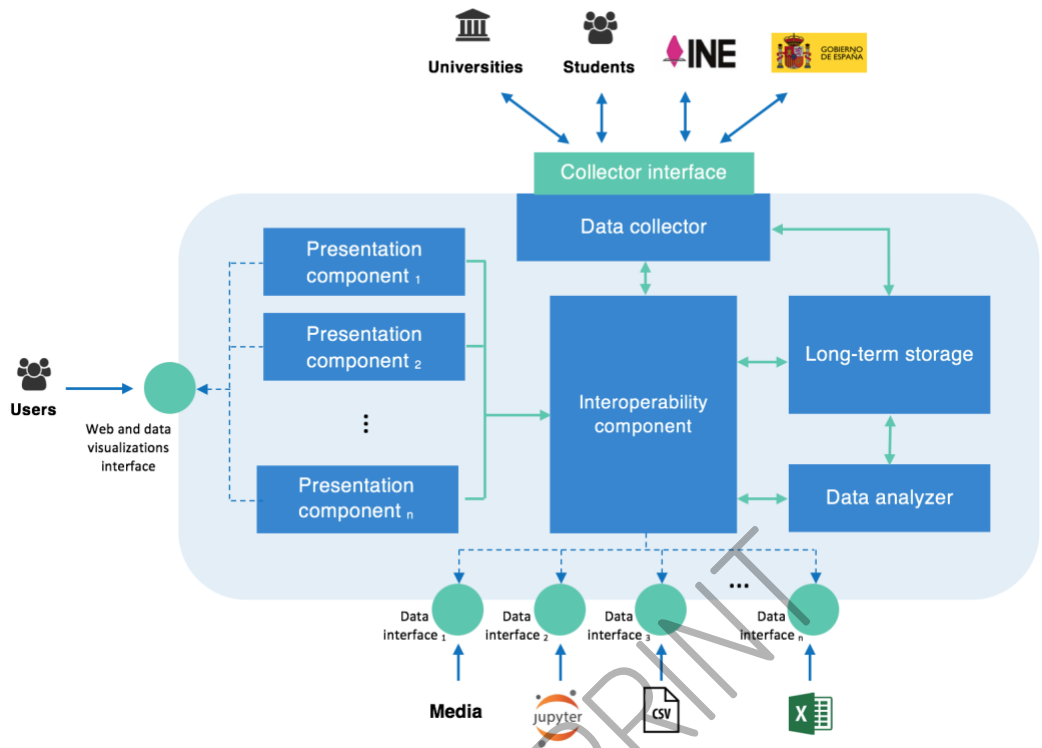

Figure 4: Overview of the OEEU's data-driven ecosystem after introducing the interoperability component (REST API).

As a consequence of these problems, the Observatory's introduced a REST API to handle the communication between both internal and external components. This REST API was designed according to the possible data flows and data requests happening within (and against) the ecosystem. With the REST API handling the interoperability, the components of the system were able to decouple from each other, converting the initial OEEU's monolithic information system into a data-driven technological ecosystem, as presented in the Fig. 4.

The REST API accomplished its goal and the ecosystem's interoperability improved, but the API stuck to the requirements at that time. This meant that any change on the data requirements or on the components of the ecosystem (i.e. when data analysts are prototyping new or different analysis or data approaches) would entail major changes on the REST API's endpoints (or even the creation of new versions of the REST API). Another limitation of the REST API was its performance. Due to the high number of metrics, variables, categories and information collections involved in the Observatory's data analysis, the number of API endpoints to retrieve them all was significant (and therefore the number of API calls needed to retrieve specific data). The REST API also lacked flexibility and reutilization, because the endpoints (their outputs) were adapted to the information needs of every component of the ecosystem. This design simplified the collaboration between the internal components, but also made it very difficult to reutilize the API endpoints for other tasks or even other types of components or software entities. Sometimes, if some component experienced a change, the whole endpoint designed for it had to be updated.

Looking to the future of this data-driven product and project, it was clear that the Observatory's REST API could evolve to a GraphQL API, since the previously outlined characteristics of this query language made it suitable and potentially beneficial for the Observatory's interoperability, scalability and maintainability needs.

3.3 The OEEU's GraphQL API

The GraphQL API for the Observatory's data-driven technological ecosystem has been implemented using Graphene (a GraphQL framework for Python, http://graphene-python.org/).

Currently, the Observatory's GraphQL API is used to create and handle the data flows within the ecosystem, and enables components to retrieve raw data about the students and statistics and metrics calculated on demand. The structure of the ecosystem remains the same (as in the Fig. 4), but now the interoperability component is based on GraphQL instead of REST.

Other significant characteristic introduced by the implementation of the GraphQL API is the fact that only one endpoint is needed (which means that only one URL is needed), against which all the queries are executed. Using the GraphQL API, components can retrieve data fields according to their data requirements. If that components' requirements change at some point, it is only necessary to modify the query of the particular component involved in the change.

While high levels of interoperability are maintained (as with the previous REST API), the independence of the components is even further enhanced. Moving the task of designing the query to the API consumer gives the clients more freedom to evolve without affecting other components.

Furthermore, the unification of the data and the structuration behind GraphQL makes possible the reutilization as well as the predictability of the results of the API calls (one of the characteristics of GraphQL [7]). This response predictability derives benefits regarding the reduction of time spending by clients parsing and restructuring data to fit their requirements.

However, although the whole data-driven ecosystem experienced an improvement, one of the main beneficiaries of the GraphQL API are the presentation components (mainly dashboards in this case), because these components are more likely to evolve and change through time. As explained in the previous subsection, visualizations are an essential tool for understanding the results of the Observatory's studies. There are a series of dashboards (supported by the ecosystem's presentation components) holding different visualizations that show analyzed data on demand. Those dashboards typically consumed data from several REST API endpoints to compose the different visualizations included in the whole screen.

With the GraphQL API providing the communication methods between the data analyzer and the presentation components, any change in the visualizations supported by any presentation component would only imply a change on the query that populates it with data. In contrast with the previous REST API, where any change on the visualizations would involve changes in the API endpoints, GraphQL promotes the decoupling and independence of the components, so any modification in a presentation component would only impact that presentation component (avoiding changes in the backend endpoints). This decoupling and independence characteristics also applies to the rest of the ecosystem's components that use the GraphQL API to handle their data flows. The other main beneficiaries of the GraphQL API are third-party components that can retrieve specific information efficiently without having to adapt themselves to the Observatory's particular endpoints' design.

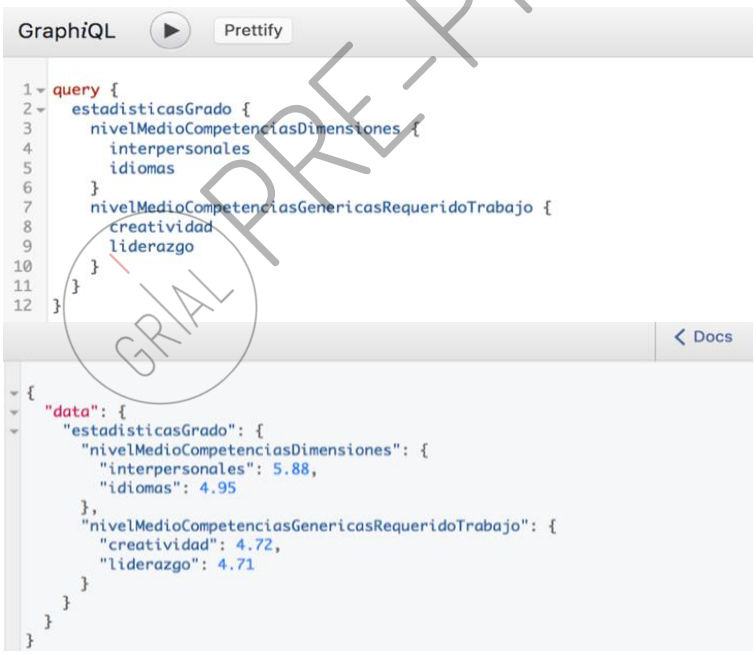

Figure 5: Example of a query (top) and a response (bottom) of The Observatory's GraphQL API (contents in Spanish).

Fig. 5 shows an example of a query to the Observatory's GraphQL API to retrieve statistics about the competencies level of the students involved in the Observatory's 2015 study (as well as the response). The example query was made via GraphiQL (https://github.com/graphql/graphiql), an interactive GraphQL IDE which allows users to explore all the possibilities of a GraphQL API. However, this query can be made through any other tool that is able to send POST requests.

4 RESULTS

The Observatory's data-driven ecosystem has mainly experienced an improvement of flexibility regarding the communication of both internal and external components, and also increased its scalability and maintainability levels, as well as the network performance. The main reason of the performance increase is the reduction of API calls to retrieve data. As pointed out before, only one endpoint is necessary, and thereby one API call is enough to query the backend.

For example, the presentation component (that manages information about the students' generic competencies levels) provides support for a total of 15 visualizations (it is currently the larger dashboard page of the Observatory's ecosystem). With the REST API as a data provider, this presentation component had to send 17 requests to different endpoints in order to retrieve all the analyzed data and visualize them. The number of API calls slowed down the data retrieval process. However, with GraphQL, a single query is issued against the backend to retrieve the same data for the entire dashboard. This supposed an increase of network performance (32.26% of average network time has been decreased) and a decrease regarding the requests' size (47.86% less of broadband loading is required).

The results summarized in Table 1 show the network average times (obtained through the network developer tool of the Mozilla Firefox browser) after 10 accesses to the generic competencies statistics dashboard and therefore, after the execution of the series of data requests needed to populate the visualizations. The analyzed information requested for this test involved data from more than 1000 students.

Table 1: Comparison of performance results between the Observatory's REST API and the GraphQL API during the load of an information dashboard. R1-R10 are the network response times obtained in each access to the dashboard.

	REST API	GraphQL API
Total request size	26.66 KB	13.90 KB
Request 1	6.54s	3.85s
Request 2	7.45s	4.17s
Request 3	5.82s	4.94s
Request 4	5.50s	3.90s
Request 5	6.14s	4.95s
Request 6	5.26s	4.16s
Request 7	5.79s	4.11s
Request 8	5.94s	3.97s
Request 9	7.06s	3.79s
Request 10	5.99s	3.77s
Average network response time	6.15s	4.16s
Improvement with GraphQL:	32.36% of network time decreased 47.86% less of broadband loading required	

Not only the average network time to retrieve all the analyzed data has been reduced by this change, but also the total size of the requests, thanks to the unification of all the previous REST API requests into a single GraphQL API request. The rest of the presentation components of the ecosystem had experienced similar network performance gains.

Although the performance gain could not be vital in some systems, the flexibility gained is vital for this type of ecosystem. If some component changes its structure or even its functionality, it is only necessary to modify the query made by the component itself to fit the new data requirements. This can save time in the development for the future regarding the data-driven ecosystem's components evolution.

The Fig. 6 shows the graphical representation of the test results.

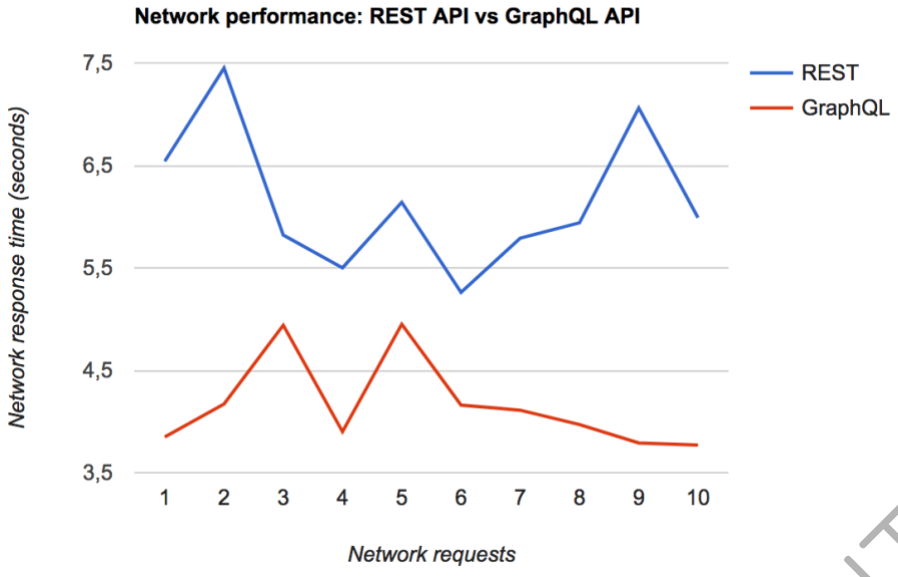

Figure 6: Graphical representation of the network response times obtained after executing the API requests.

5 DISCUSSION

Due to the improvement of the Observatory's ecosystem interoperability, some issues associated to their technical challenges have been solved in the past with the REST API. Although the interoperability levels provided by the REST API implemented were enough at the time, the potential evolution of the requirements and the components of the Observatory asked for more flexibility and scalability levels.

The introduction of the GraphQL API to the Observatory's data-driven technological ecosystem provided flexibility, scalability and maintainability regarding its components' evolution. Also, the network time to retrieve the data has been reduced, which is another important benefit giving the significant amount of data handled.

Scalability and flexibility are characteristics that the previous REST API lacked. These characteristics are crucial for this data-driven ecosystem considering that the Observatory's fields of study (employment and university employability) are continuously evolving and the data amount gathered to analyze is continuously increasing.

With the REST API, the backend developers had to design and specify the data responses for every endpoint of the API before any client could use it. But with the GraphQL API, developers only need to specify the ecosystem's data objects (and their fields) available and even filters or operations, and then the GraphQL framework chosen will handle the clients' queries.

This approach makes the addition and modification of data objects (and fields) quasi-trivial, because only the GraphQL entities are involved in the change, in contrast with REST APIs', where a series of endpoints could be affected by the modification of the data entities.

It is task of the clients (data consumers and prosumers) to design their queries based on their requirements and the data available on the backend, and not of the backend to design the endpoints based on the clients' data requirements. The backend makes available their data resources, and the clients decide how to consume it.

But not only the internal components of the data-driven ecosystem are beneficed by the GraphQL approach; external components (whose data requirements are out of the Observatory's control) now have a flexible communication method to connect themselves to the internal ecosystem's components.

GraphQL has also resolved the issue regarding the specificity of the REST endpoints; most of the endpoints of the Observatory's previous REST API were designed specifically for its internal components, so, although external components could access these endpoints, the results of their request could not fit the requirements to accomplish their purposes. The GraphQL API gives freedom to external components to design their own queries and promotes the exploitation of the knowledge obtained by the Observatory. This also could represent issues regarding the security: The GraphQL API should include object permission and access control depending the client that is using the API.

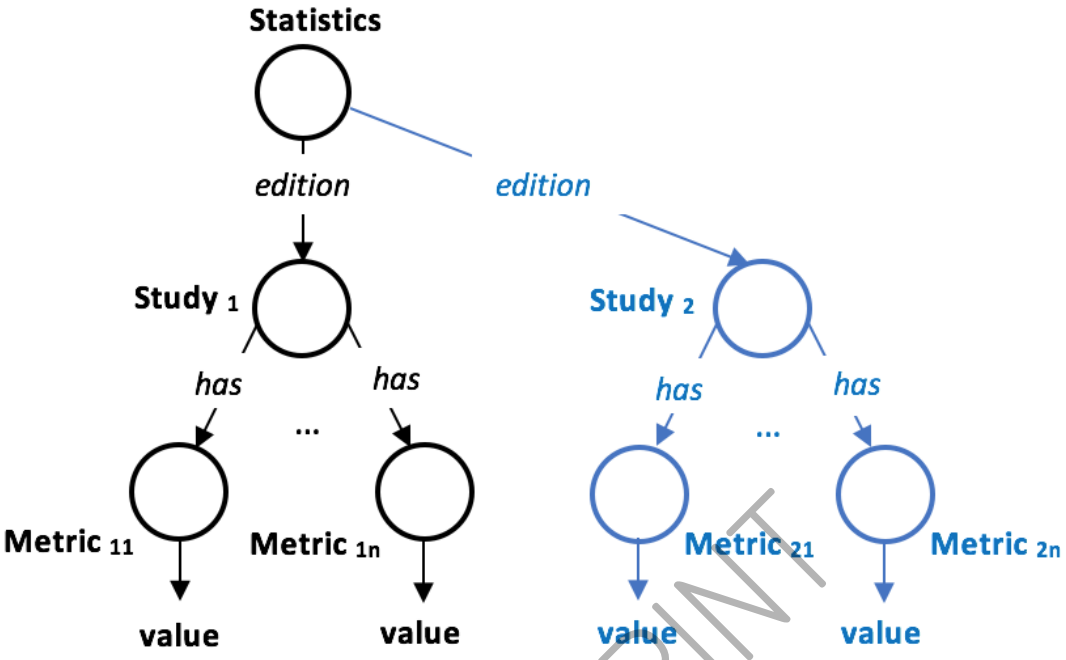

Figure 7: Visual structure of the OEEU's statistics using GraphQL. The addition of new metrics and fields implies the creation of new nodes and new relations between the already existent nodes.

Finally, the data graph-structuration provided by GraphQL gives the Observatory a more scalable and organized way to retrieve and analyze the collected data.

For instance, the Observatory needs a data entity to represent the metrics' values derived from their collected data in order to ease the reaching of insights about employment and employability. With GraphQL, this data entity can be represented by a root node symbolizing the whole set of statistics generated by the Observatory. However, the Observatory is making a series of study editions through time, and the metrics' values vary depending on the study edition. To address that problem, another node symbolizing the study edition can be attached to the root node. Then, it is only necessary to define the metrics and, again, attach them to the graph as leaf nodes of the specific study edition.

This approach has two benefits. The first benefit is that metrics are organized by the study edition, making the retrieval more intuitive for the clients. The second benefit is the increase of scalability regarding the Observatory's data; new data entities representing new editions of the Observatory's studies can join the initial root node of the data graph, minimizing the impact of the data's evolution. This structure is conceptually showed in Fig. 7: the black colored elements can be seen as the initial graph created for the first edition of the Observatory's study. Then, the following editions only need to join the statistics root node, creating a clean structure to browse and retrieve data.

This structure also allows the retrieval of data from different studies' editions through one unique request, which simplifies the analysis of the evolution of the metrics' values through time.

There are, though, some concerns to keep in mind derived from the GraphQL API's implementation. The hierarchical structure of the queries can suppose a threat: the query of deep nested relations could end up in a denial of service attack and consume all the resources of the backend [15].

It is important to keep this kind of attacks in mind if the GraphQL API continues growing, as well as the authentication and authorization methods (as previously pointed out in the case of the Observatory).

Although the GraphQL API has reported significant benefits, its introduction to the ecosystem did not imply the shutdown of the previous REST API. The approach chosen uses the GraphQL API for internal and external components' communication, which have particular technical requirements, but other users might not have these scalability, flexibility and performance needs.

REST APIs are simple, extended and don't require the design of a particular query, so they could be useful for general users of the Observatory

6 CONCLUSIONS

GraphQL can be seen as a powerful solution to increase the interoperability of data-driven and data-intensive (eco)systems because it provides high levels of flexibility, which help to support changing requirements along time.

The use of this query language also comes with an increase of performance due to the reduction of the number of requests, as well as higher levels of scalability and maintainability thanks to its "graph nature" [15].

As data-driven and data-intensive (eco)systems are composed by continuously evolving components (which have to stick to changing requirements), their scalability, flexibility and performance needs are crucial. These needs are what make the GraphQL approach suitable for these type of (eco)systems.

With GraphQL, the concept of data-as-a-service (DaaS) [16] is more authentic; data is provided on demand and clients can specify the structure, filters or even operations for the data retrieved.

However, the arising of GraphQL does not have to mean that REST APIs are going to disappear. Although the benefits derived from the use of GraphQL could make this language preferable over REST [15], this last protocol remains a suitable and simple solution for lots of systems that doesn't have critical interoperability and flexibility needs.

ACKNOWLEDGMENTS

The research leading to these results has received funding from "la Caixa" Foundation. Also, the author Juan Cruz-Benito would like to thank the European Social Fund and the *Consejería de Educación* of the *Junta de Castilla y León* (Spain) for funding his predoctoral fellow contract. This work has been partially funded by the Spanish Government Ministry of Economy and Competitiveness throughout the DEFINES project (Ref. TIN2016-80172-R).

REFERENCES

[1] Dj Patil and Hilary Mason. 2015. *Data Driven*. " O'Reilly Media, Inc.".

[2] A. Vázquez-Ingelmo, J. Cruz-Benito, and F. J. García-Peñalvo. In press. Scaffolding the OEEU's Data-Driven Ecosystem to Analyze the Employability of Spanish Graduates. In *Global Implications of Emerging Technology Trends*, F.J. García-Peñalvo Ed. IGI Global.

[3] A. García-Holgado and F. J. García-Peñalvo. 2016. Architectural pattern to improve the definition and implementation of eLearning ecosystems. *Science of Computer Programming* 129, 20-34. DOI:http://dx.doi.org/10.1016/j.scico.2016.03.010.

[4] F. J Garcia-Peñalvo and A. Garcia-Holgado. 2016. *Open Source Solutions for Knowledge Management and Technological Ecosystems*. IGI Global.

[5] R. T Fielding and R. N. Taylor. 2002. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)* 2, 2, 115-150.

[6] M Faassen. 2015. GraphQL and REST, Secret Weblog.

[7] Facebook. 2016. GraphQL.

[8] S. Taylor. 2017. React, Relay and GraphQL: Under the Hood of the Times Website Redesign. In *Times Open*.

[9] F. Michavila, M. Martín-González, J. M Martínez, F. J. García-Peñalvo, and J. Cruz-Benito. 2015. Analyzing the employability and employment factors of graduate students in Spain: The OEEU Information System. In *Proceedings of the 3rd International Conference on Technological Ecosystems for Enhancing Multiculturality* ACM, 277-283.

[10] F, Michavila, J. M. Martínez, M. Martín-González, F. J. García-Peñalvo, and Juan Cruz-Benito. 2016. Barómetro de Empleabilidad y Empleo de los Universitarios en España, 2015 (Primer informe de resultados).

[11] J. Bichsel. 2012. *Analytics in higher education: Benefits, barriers, progress, and recommendations*. EDUCAUSE Center for Applied Research.

[12] Á. Fidalgo-Blanco, M. L. Sein-Echaluce, and F. J. García-Peñalvo. 2014. Knowledge Spirals in Higher Education Teaching Innovation. *International Journal of Knowledge Management* 10, 4, 16-37. DOI:http://dx.doi.org/10.4018/ijkm.2014100102.

[13] Á. Fidalgo-Blanco, M. L. Sein-Echaluce, and F. J. García-Peñalvo. 2015. Epistemological and ontological spirals: From individual experience in educational innovation to the organisational knowledge in the university sector. *Program: Electronic library and information systems* 49, 3, 266-288. DOI:http://dx.doi.org/10.1108/PROG-06-2014-0033.

[14] A. García-Holgado, J. Cruz-Benito, and F. J. García-Peñalvo. 2015. Analysis of knowledge management experiences in spanish public administration. In *Proceedings of the 3rd International Conference on Technological Ecosystems for Enhancing Multiculturality* ACM, 189-193.

[15] S. Buna. 2017. Rest APIs are REST-in-Peace APIs. Long Live GraphQL.

[16] O. Terzo, P. Ruiu, E. Bucci, and F. Xhafa. 2013. Data as a service (DaaS) for sharing and processing of large data collections in the cloud. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on* IEEE, 475-480.