



REST vs. GraphQL – Building APIs for abstract applications

Simo Öysti

Bachelor's thesis

December 2021

Information and Communication Technologies

Bachelor's Degree Programme in Information and Communication Technology

Öysti, Simo

REST vs. GraphQL – Building APIs for abstract applications

Jyväskylä: JAMK University of Applied Sciences, December 2021, 67 pages

Information and Communications. Degree Programme in Information and Communication Technology. Bachelor's thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

APIs can be found everywhere in today's digital ecosystems. There exists a wide variety of clients that utilize them, ranging from mobile or web applications to the software of physical devices, even robot vacuums or smart lighting appliances. Through the global digital transformation, APIs are utilized increasingly, allowing applications to readily communicate with one another. Due to the continuously increasing demand for such integrations, APIs have become a standardized part of modern-day application development.

A need for enabling such integrations arose at the University of Jyväskylä, where the current research information system needed a flexible solution to enable other parties to utilize the data within. Main requirements for the product were good permissions management, no over-fetching of data, and comprehensive logging. Considering the increasing need and popularity of APIs, interest was also taken in finding and implementing an abstract enough solution, so that any other applications could directly utilize the same approach in future integrations.

The solution, aimed to cater to the needs of as many applications at once as possible, was a read-only API utilizing GraphQL. The product was implemented on top of an on-premises RHEL 8 server in a virtual environment. It consisted of a Hasura GraphQL server running within Podman containers, having an additional Apache reverse proxy on the front. An external PostgreSQL database was connected to Hasura to function as the primary data source. Additionally, an authentication server was set up with the help of Node.js to provide users with authentication and thus enable role-based authorization.

The resulting product of the research project became a fully functional proof of concept, requiring only further minor configurations to fully fit the specific needs of the employer. At the same time, the implementation remained abstract enough so that it could be directly applied to other applications, without any existing knowledge of the technologies related to the research project.

The purpose of the research project was to investigate relevant technologies, based on which a new practical API for the University of Jyväskylä was to be created to support the increasing demand for application integrations. These objectives were achieved, as the final product was something that could be immediately utilized to some extent. Optional future improvements were also highlighted, based on which the employer could easily further expand functionality according to future needs.

Keywords/tags (subjects)

API, Application programming interface, DB, database, GraphQL, Hasura, REST, Node.js

Miscellaneous (Confidential information)

Öysti, Simo

REST vs. GraphQL – Ohjelmointirajapintojen rakentaminen abstrakteja sovelluksia varten

Jyväskylä: Jyväskylän ammattikorkeakoulu. Joulukuu 2021, 67 sivua

Tietojenkäsittely ja tietoliikenne. Tieto- ja viestintäteknikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: englanti

Verkkojulkaisulupa myönnetty: kyllä

Tiivistelmä

Ohjelmointirajapinnat ovat vahvasti läsnä nykypäivän digitaalisissa ekosysteemeissä. Niitä hyödyntäviä sovelluksia on laajalti, aina mobiili- ja web-sovelluksista fyysisten laitteiden, jopa robotti-imurien ja älyvalaisimien ohjelmistoihin. Ohjelmointirajapintoja hyödynnetään enenevässä määrin maailmanlaajuisen digitaalisen transformaation myötä, mikä lisää sovellusten valmiutta keskustella keskenään.

Ohjelmistointegraatioiden jatkuvasti lisääntyvän kysynnän ansiosta ohjelmointirajapinnoista on tullut standardisoitu osa nykypäivän sovelluskehitystä.

Tarve ohjelmistointegraatiolle Jyväskylän yliopistolla nousi esille, kun tutkimustietojärjestelmä tarvitsi joustavan ratkaisun mahdollistaakseen datansa hyödyntämisen muille osapuolille. Tärkeimmät vaatimukset tuotteelle olivat hyvä käyttöoikeuksienhallinta, liiallisen datan kerralla hakemisen välttäminen ja kattava lokitus. Huomioiden ohjelmointirajapintojen kasvavan kysynnän, haluttiin lisäksi löytää ja toteuttaa tarpeeksi hyvin abstraktoitu ratkaisu, jotta mitkä tahansa muut sovellukset voisivat suoraan hyödyntää samaa lähestymistapaa tulevilla integraatioissa.

Ratkaisu, joka tähdättiin vastaamaan niin monen sovelluksen tarpeisiin kuin mahdollista, oli GraphQL:ää hyödyntävä kirjoitusuojattu ohjelmointirajapinta. Tuote toteutettiin paikalliselle RHEL 8-palvelimelle virtuaaliympäristössä. Se sisälsi Hasura GraphQL-palvelimen, joka toimi Podman-konteissa ja sen edessä toimi lisäksi käänteisenä välityspalvelimena Apache. Ulkoinen PostgreSQL-tietokanta yhdistettiin Hasuraan toimia-akseen pääasiallisena tietolähteenä. Lisäksi autentikaatiopalvelin pystytettiin Node.js:n avulla tarjoamaan käyttäjille autentikaatiota ja siten mahdollistamaan roolikohtaisen auktorisoinnin.

Tutkimusprojektin lopputuotoksena syntyi kaikin puolin toimiva tutkimusprototyyppi, joka vaatisi ainoastaan vähäistä konfiguraatiota sopiakseen täysin toimeksiantajan tarkempiin tarpeisiin. Samalla toteutus pyysi niin abstraktina, että sitä voitaisiin suoraan käyttää muiden sovellusten kanssa ilman olemassaolevaa tietämystä tutkimusprojektiin liitetyistä teknologioista.

Tutkimusprojektin tavoitteena oli tutkia olennaisia teknologioita, joiden perusteella toteutettaisiin uusi käytännöllinen API Jyväskylän yliopistolle tukemaan ohjelmistointegraatioiden lisääntyvää tarvetta. Nämä tavoitteet saavutettiin, sillä valmista tuotetta voitaisiin välittömästi hyödyntää tietyssä määrin. Myös vaihtoehtoisia kehityskohteita korostettiin, joiden pohjalta toimeksiantaja voisi helposti edelleen laajentaa toiminnallisuutta tulevien tarpeiden mukaiseksi.

Avainsanat (asiasanat)

API, ohjelmointirajapinta, tietokanta, GraphQL, Hasura, REST, Node.js

Muut tiedot (salassa pidettävät liitteet)

Table of Contents

Acronyms	5
1 Introduction	6
1.1 Subject.....	6
1.2 Project goals.....	6
1.3 Scope	7
1.4 Research questions	8
1.5 Research methodology	8
2 Project environment	9
2.1 Application	9
2.2 Data model.....	10
2.2.1 Official standards.....	10
2.2.2 Converis	13
2.3 Infrastructure	14
3 Application programming interface	15
3.1 Definition.....	15
3.2 Clients.....	16
3.3 Behavior	17
4 REST	17
4.1 Introduction.....	17
4.2 The 5 principles of REST	19
4.2.1 Client-server architecture.....	19
4.2.2 Statelessness.....	20
4.2.3 Uniform interface	20
4.2.4 Caching.....	21
4.2.5 Layered architecture.....	22
4.3 Versioning.....	22
5 GraphQL	23
5.1 Introduction.....	23
5.2 Schema	23
5.3 Resolvers	24
5.4 Queries	25
5.5 Versioning.....	26
5.6 Authentication and authorization.....	26

6	Choosing the right approach	28
6.1	Unified APIs	28
6.2	Project specifications	28
6.3	Performance.....	29
6.4	Complex data models.....	29
6.5	Users.....	30
6.6	Conclusion	31
7	Hasura.....	32
7.1	Introduction.....	32
7.2	Features.....	33
7.2.1	Automation	33
7.2.2	Query performance analysis.....	33
7.2.3	Authentication	34
7.2.4	Authorization	34
7.2.5	Cloud	36
7.3	Suitability for the project	36
8	Implementation.....	37
8.1	Prerequisites.....	37
8.2	Hasura installation	38
8.3	Configuration.....	42
8.3.1	Data source	42
8.3.2	Data definitions.....	43
8.3.3	Queries and API endpoints	45
8.3.4	Authentication	49
8.3.5	Authorization / Access control	53
8.3.6	Logging.....	55
9	Results.....	56
10	Conclusions	59
10.1	The chosen tools	59
10.2	Challenges	60
10.3	Future expansion.....	61
	References	63
	Appendices	67
	Appendix 1. The final GetPublication query	67

Figures

Figure 1. Entities and their links in Converis.....	9
Figure 2. Top-level configuration interface of Converis	10
Figure 3. Old CERIF 2000 data model visualization.....	11
Figure 4. CERIF parts relevant for OpenAIRE	12
Figure 5. VIRTAs-OpenAIRE integration.....	13
Figure 6. Examples of derived relations in Converis	14
Figure 7. Visualization of both sides of the API	16
Figure 8. Example of a common REST architecture diagram.....	19
Figure 9. HTTP caching	22
Figure 10. Sample schema with object and query types	24
Figure 11. Sample schema with object, query and mutation types	24
Figure 12. Resolvers with hardcoded return values	25
Figure 13. GraphQL query and response in JSON	26
Figure 14. Access control layers.....	27
Figure 15. Authentication and authorization flow.....	28
Figure 16. Comparison of GraphQL and REST.....	31
Figure 17. Generated SQL and execution plan on Hasura web console.....	34
Figure 18. Access control rule creation in Hasura	35
Figure 19. Modified docker-compose file for bringing up Hasura and its internal database.....	41
Figure 20. Hasura web console	41
Figure 21. Adding a data source to Hasura through an environment variable	42
Figure 22. Converis Database relationships.....	44
Figure 23. GetPublication v1, querying data based on relations.....	45
Figure 24. GetPublication v2, querying values through several relation paths.....	46
Figure 25. GetPublication v3, introducing variables	47
Figure 26. Wrapping a whole GraphQL query within a simple HTTP endpoint.....	47
Figure 27. The GetPublication HTTP endpoint in action.....	48
Figure 28. Testing new user signup with Passport.js	52
Figure 29. Testing Passport.js token retrieval through the /login route	53
Figure 30. Granting the role "user" unrestricted SELECT access to a single table	54
Figure 31. Unauthorized POST request from an anonymous user	54
Figure 32. Authorized POST request from an authenticated user	55
Figure 33. Example entry From Hasura's logs (JSON)	56

Figure 34. Node authentication server's response handler function 61

Acronyms

API	Application programming interface
CERIF	Common European Research Information Format
CRIS	Current research information system
CRUD	Create, Read, Update, Delete
DB	Database
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of things
IRI	Internationalized Resource Identifier
JS	JavaScript
JSON	JavaScript Object Notation
JWT	JSON Web Token
OpenAIRE	Open Access Infrastructure for Research in Europe
OS	Operating System
REST	Representational state transfer
RHEL	Red Hat Enterprise Linux
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

1.1 Subject

The purpose of this thesis is to present the design and implementation processes of an API, built to function independently regardless of the underlying applications and their potentially complex data models, with the ease of deployment and re-configuration in mind. Various aspects of APIs and their development practices were studied to accomplish this. This subject was determined from a development task assigned by the University of Jyväskylä, as their current research information system (CRIS) could benefit from such interface.

The research problem was found to be the way that the current API of the CRIS application was built. The API was created as an improvised solution to a specific requirement, without much development resources spent on the bigger picture of what the API could be at its best, such as supporting easy scalability, reconfiguration for potential future updates, or additional ways in which other parties could be utilizing the data of the CRIS application. Considering how much a system that is being widely queried by other parties can benefit from a reliable, robust API, replacing such ad hoc solutions most certainly benefits everyone in the end (Brooks, G. 2013).

1.2 Project goals

One of the main areas of improvement that was highlighted was the simplification of the data model, as the old API solution could return various nonessential technical attributes, or even hidden, potentially confidential fields in its responses, any of which the client should not be receiving most of the time. There also existed no authentication method, which would support the differentiating of clients or allow the configuration of client-specific rights within the API. Instead, any requests sent to the API were executed with administrator privileges. This kind of implementation, by nature, limits the scope of the audience that could even have any kind of access to the API.

The main goal of this research project was to create a solution, which would not only improve on the existing API, but also support the deployment of the API on top of an existing, abstract application. This meant that the research would remain focused on finding a way to deploy and configure

the API on different kinds of existing applications, instead of tailoring it for, or possibly even integrating it into the CRIS application in question. This would not only let other systems potentially adapt the same solution, but also allow the API to be easily re-configured on the server-side, in the case that any major changes occur. In an ideal scenario, any changes to the background application or its data structure would cause no visible changes to the externally visible API specification and thus require no actions from any of the clients already utilizing the API.

Having an API to maintain does increase the workload of the developers on the server side when changes in some form eventually do occur. However, as any of the planned clients for this internal API were also going to be using resources of the employer, with or without the API, and the number of clients for an application such as the CRIS could increase as well in the long run, reducing the workload for the clients of the API was a priority as well. The API was therefore to be designed in a way that both gives the client developers more freedom in how the API of the CRIS application is accessed, while also minimizing the need for client-sided development work and changes during possible future updates to the background application.

1.3 Scope

For the sake of keeping focus on abstraction, the scope of this thesis was restricted to consider a read-only API. Not only is a read-only API sufficient for the long-term needs of the employer, but modifying data in an abstract, existing application, poses its own separate challenges considering how differently data changes could be managed in different systems. An abstracted API deployment might likely be set up to interact with the backend database directly. Direct data writes to the DB would then not properly trigger scripted on-data-changed events within the application, to give an example of a challenge that would rise from implementing a CRUD API with the specific CRIS application in question.

A proper authentication method, however, was required so that client-specific data read permissions could be managed, which in turn would allow the benefits of the API to be opened to different groups of users. A complete logging functionality for the API was also mandatory, as any existing logging functionalities of the application would be effectively bypassed, if interacting with the DB directly.

1.4 Research questions

Based on the research problem and goals, the main research question was determined to be “How to design and implement a read-only API solution well suited for an abstract, existing application?”

Additionally, the main question was split into the following sub questions:

- How easy is it to deploy the final product or make changes to it?
- Is the final product a considerable API option for other applications?
- How far could the API be reasonably abstracted?
- Is the amount of work required from the API clients’ developers reasonable?

The research in this thesis focuses on studying the industry standards relevant to the application in question, as well as finding best practices related to API development in general. This results in discovery of the required information, based on which the final product is developed and finally implemented for the employer.

1.5 Research methodology

Determining the main research method is a starting point for research. Different research methods exist for different situations, and they all share similarities. The chosen research method designates which methods and tools will be used, and it should be appropriate for the research questions. (Kananen 2015, 29.)

Qualitative research is the method used in this thesis. Qualitative research strategy aims to expand the overall understanding of the problem, to grant in-depth insights on topics that are not understood well enough (Bryman 2012, 380-384). During the project at hand, the understanding of various API development techniques was expanded by studying them in detail, after which a suitable product was developed based on the findings. Some characteristics were also used from the constructive research method, which is a systematic approach used to define issues and solve them by improving on the existing. (Lukka 2003, 85.)

2 Project environment

2.1 Application

Converis™ is the application upon which the API would be built. It is a CRIS system developed by Clarivate, built to integrate the management of research-related information with the workflows of the business. It allows this by making it possible to join multiple sources of data, both internal and external, such as repositories, libraries, and databases among other systems of the institution and creating complex workflows for managing this data. Converis is designed to manage many types of data objects, called “entities”, as well as relations between them, called “link entities”, which is visualized on Figure 1. (Converis N.d.)

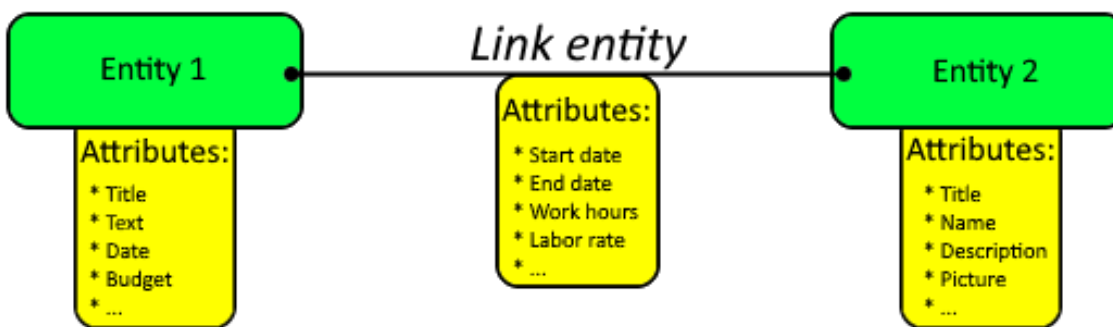


Figure 1. Entities and their links in Converis. Not only entities hold attributes, but the relationships between them do as well.

Typical examples of such entities would be a person, organization, publication, project, project application or patent (Converis N.d.). Any other kinds of entities could also be manually created and workflows for them customized by the administrator of the system, as the data model and workflows of the system are very detailed and extremely well configurable from the user interface of Converis (see Figure 2).

Web of Science InCites Journal Citation Reports Essential Science Indicators EndNote

Converis Q Search Help Config Admin: Öysti, Simo

Dashboard > Configuration

Configuration

- System Resources
 - Choice Groups (177)
 - Choice Group Value (1546)
 - Constants (140)
 - Timers (17)
- Data Model
 - Entities (65)
 - Link Entities (418)
 - Attributes (4696)
 - Link Entities Chains (60)
 - Validation Overview
- Template management
 - Entities (504 + 123 + 17)
 - Template migration utility
 - Reports (11)
 - CVs (5)
 - PDF Template
 - Archiving
 - Registration Settings
- Rights Management
 - User Roles (28)
 - Role Mapping for Research Analytics
 - Create Rights (140)
 - Dynamic templates per user role (5)
 - Basic Rights (318)
 - Special Rights (543)
 - Workflow Matrix (192)
- Module Settings
 - Left Navigation Bar (23)
 - Visibility Rules (65)
 - Entity search (607)
 - Header Search (221)
 - Statistical Charts (9)
 - Deduplication Settings
- Function settings
 - Save As (33)
 - Filter Settings (272)
 - CSL (102)
 - Copy Relation Rule (11)
 - Export config
 - Claimings (0)
 - Batch relate
 - Batch status

Figure 2. Top-level configuration interface of Converis

2.2 Data model

2.2.1 Official standards

CERIF

CERIF is a conceptual model for describing the research domain, history of which points back all the way to the late 1970s. CERIF has been recommended as the official standard for member states of the EU whenever dealing with research information systems, as described by the European Commission (1991) in their official EU Recommendation to Member States. (EuroCRIS 2014)

In early 2000, the care and custody of CERIF was handed over by the European Union to euroCRIS, which is a non-profit organization registered in the Netherlands and manages CERIF to this day. At the time, the CERIF 2000 model (Figure 3) had already become a significant improvement in describing the research domain, in comparison to the releases from the 1990s. The CERIF model has

grown since then through history, while having been influenced by technological developments. CERIF has undergone major upgrades such as one regarding normalization in 2006, and the introduction of the Semantic Layer in 2008. The CERIF 1.4, released in early 2012, also brought in an embedded XML exchange format which has since become extremely popular. (ibid.)

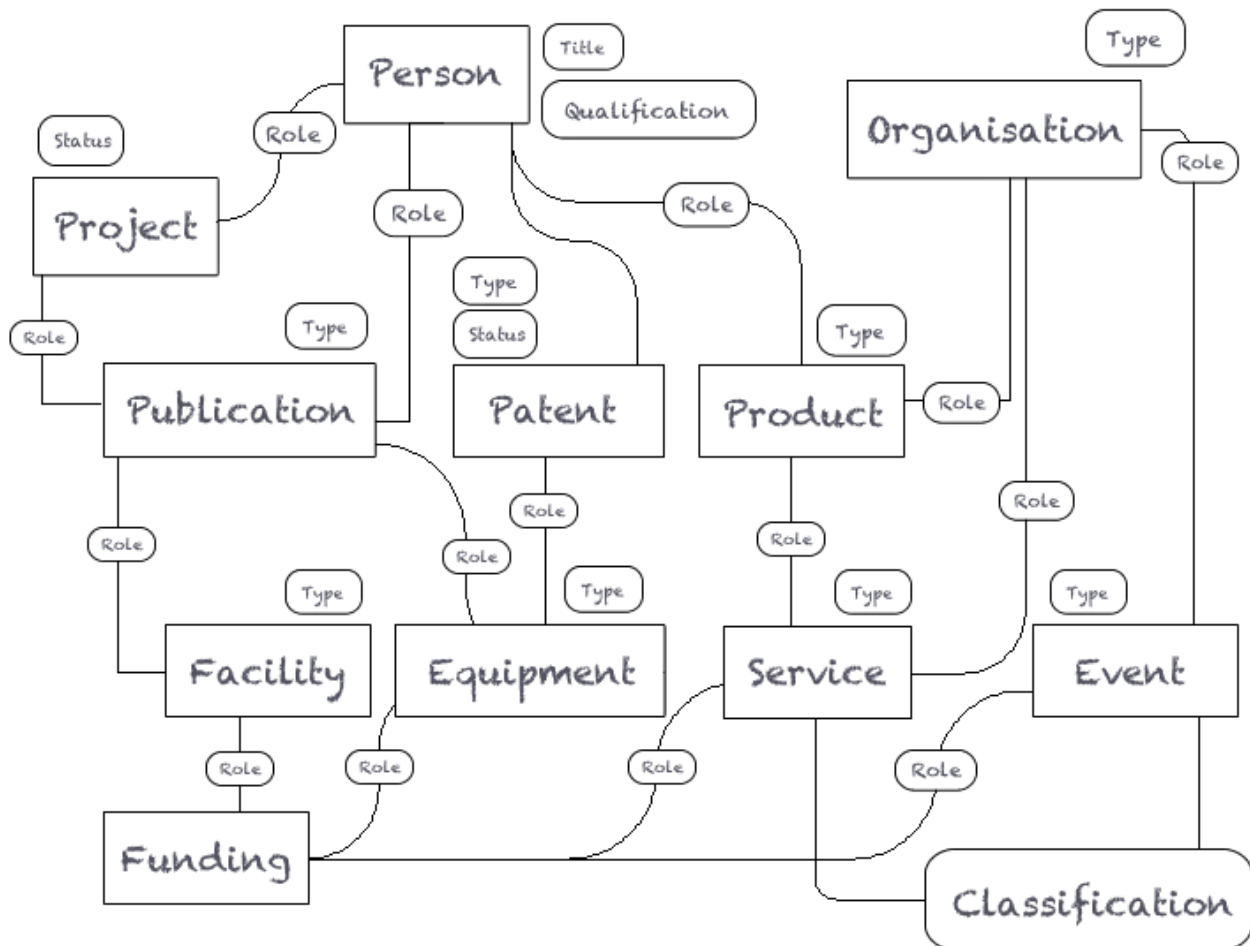


Figure 3. Old CERIF 2000 data model visualization (EuroCRIS 2014)

OpenAIRE

OpenAIRE is a European project with the purpose of supporting Open Science, launched in 2010 by the European Commission (Open Science EU 2020). OpenAIRE functions as both a technical infrastructure responsible for the overall management, analysis, manipulation, provision, monitoring and cross-linking of all research outcomes as well as a network of dedicated Open Science professionals who can promote and provide training on Open Science (OpenAIRE N.d.).

While CERIF is a comprehensive model for the research domain, only some parts of model's information are relevant for OpenAIRE, as seen on Figure 4. OpenAIRE also provides guidelines which define in detail the XML data elements for this information, to standardize the exchange of data between individual CRIS systems and the OpenAIRE infrastructure. (ibid.)

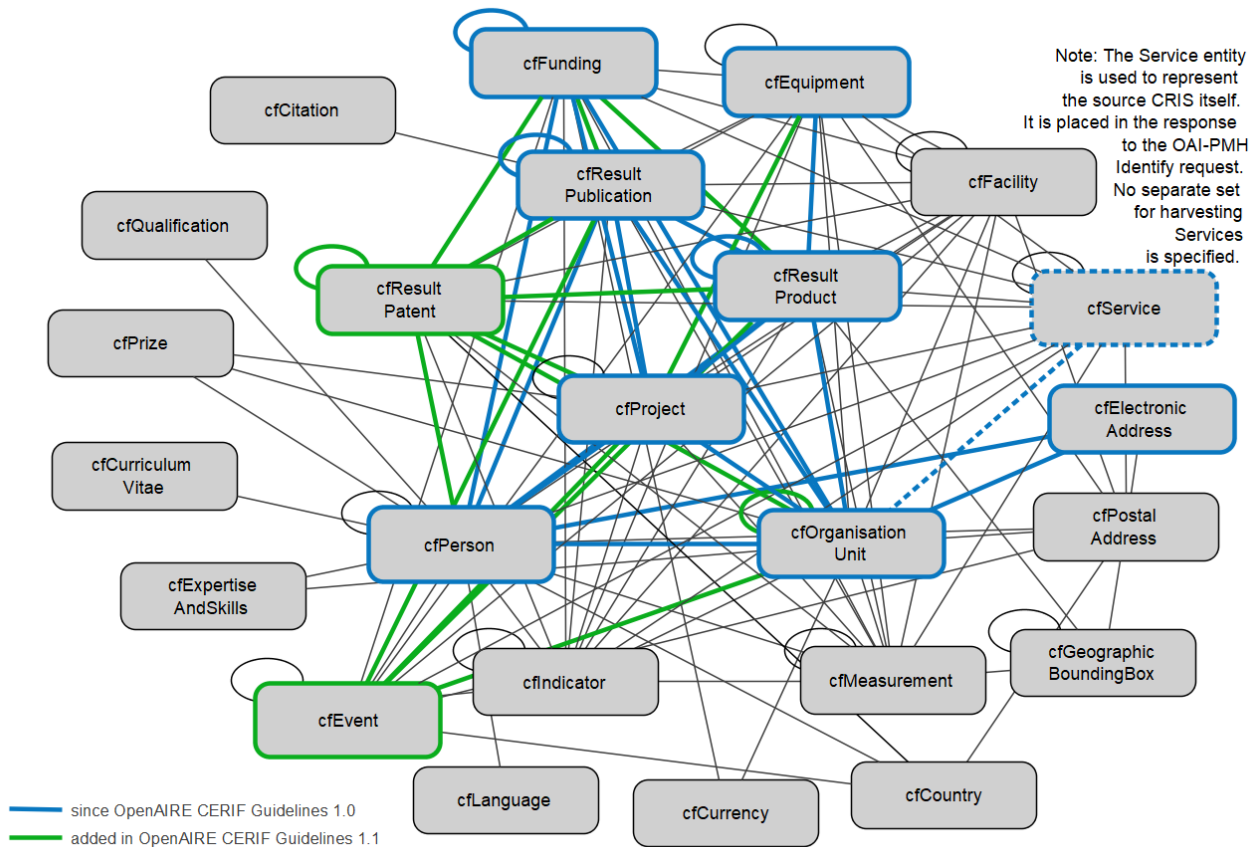


Figure 4. CERIF parts relevant for OpenAIRE (OpenAIRE N.d.)

VIRTA

The VIRTA Publication Information Service, operated by CSC (also known as IT Center for Science Ltd.), holds research information on publications produced by Finnish institutions that perform research. This includes the universities and polytechnics of Finland, among other research organizations. The VIRTA service thus acts as a source of consistent and up-to-date information on Finnish scientific publications. Information stored in VIRTA may be browsed freely by anyone at <https://research.fi/> (CSC N.d.)

The Ministry of Education and Culture in Finland performs collection of Finnish publication data annually, which is accomplished through the VIRTAs Publication Information Service. VIRTAs therefore plays a big role in the domain of Finnish research by bringing publication information into one place, from which it can then be accessed by a variety of different systems and services, effectively also reducing the administrative workload of researchers themselves. (ibid.)

As seen on Figure 5, there have been issues with the compatibility between commercial CRIS platforms used in Finland and the OpenAIRE infrastructure. To solve this, OpenAIRE and VIRTAs began a collaborative effort in 2018 to bring Finnish publication metadata to OpenAIRE, increasing not only the quality of the metadata but also the visibility of it at a global level. (Nikkanen & Schirrwagen 2019)



((())) Challenge & scenario

In Finland, none of the commercial CRIS platforms are compatible with the OpenAIRE aggregation requirements, nor do the repositories cover all the research results available from academic institutions in Finland.



Service use case



Key Solution & implementation

The aim of the integration is to provide metadata from Finnish publications of the national aggregator, VIRTAs, to OpenAIRE, thereby improving the quality of the metadata and the visibility of Finnish research results at European and international level from a single access point - making use of OpenAIRE Guidelines for CRIS Managers.

Figure 5. VIRTAs-OpenAIRE integration (Nikkanen & Schirrwagen 2019)

2.2.2 Converis

As stated, commercial CRIS platforms in Finland tend to be incompatible with the requirements of OpenAIRE. This also applies to Converis, which differs from the previously mentioned standards in

various ways. As VIRTAs takes care of the annual publication information collection and integration with OpenAIRE, the metadata is still ultimately brought from Converis to OpenAIRE.

While Converis differs from the standards, it shines with the flexibility of its data model and especially with how it allows detailed configuration of the workflows, based on the complex and even deep chains of derived links between entities. A few good examples of such derived relations are seen on Figure 6, where a link entity does not exist between two persons, but their relation can still be derived from involvement in the same projects or publications. This could even be applied to organizations 1 and 2, which could be associated with each other through the links to persons who have further links to common projects.

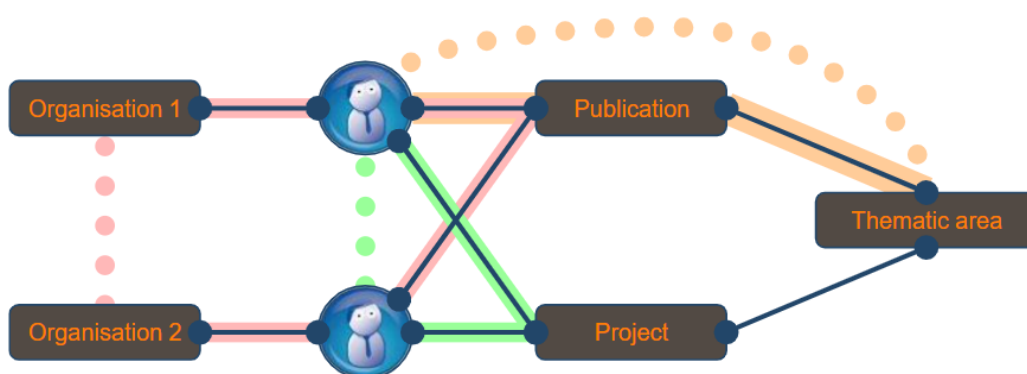


Figure 6. Examples of derived relations in Converis

2.3 Infrastructure

The installation of the Converis application which the API would be interacting with, exists on virtual RHEL servers. While there are many components and various functionalities to Converis, existing not only on a single virtual machine but across multiple servers, the only relevant technical part of the Converis installation regarding the product of the thesis was the database. For the database, Converis uses PostgreSQL, an open-source object-relational database system that has been under active development for over 30 years (Momjian 2001). While Converis uses PostgreSQL, other applications could be using other kinds of databases, which had to be taken into consideration when coming up with an abstract API solution. The relevance of the underlying database infrastructure to the API should therefore be minimal.

3 Application programming interface

3.1 Definition

One way to look at an API is to consider it as a middleman between a system and outside users wishing to utilize it, who do not necessarily possess inside knowledge of the system or even have any kind of access rights to it. Generally, an API would selectively disclose data to the users of another abstract application, or sometimes even allow them to perform actions within the system on which the API has been built (Read-only vs. CRUD APIs), without requiring them to know any details of what's happening behind the API. The purpose of an API is therefore to abstract the details of each system to bring them closer together (Pearlman 2016). This abstraction ultimately allows clients to utilize the hidden backend system(s) in a very simple manner, only requiring knowledge of the technical specification of the API itself, which again is specifically designed for the outside clients to use (ibid.).

The API is everything that the clients will see and work with, as explained on Figure 7. When a client is interacting with an API, internally the API is working with other systems in ways that the user does not know, nor do they need to. To get its work done, the API might even communicate with any number of other internal APIs, which again might do the same, and so forth, becoming a "chain" of APIs. This, however, would not concern the outside user at all, as all the work would happen in the shadows behind a single API, and the client would only see the result which is eventually sent back to them by this "middleman".

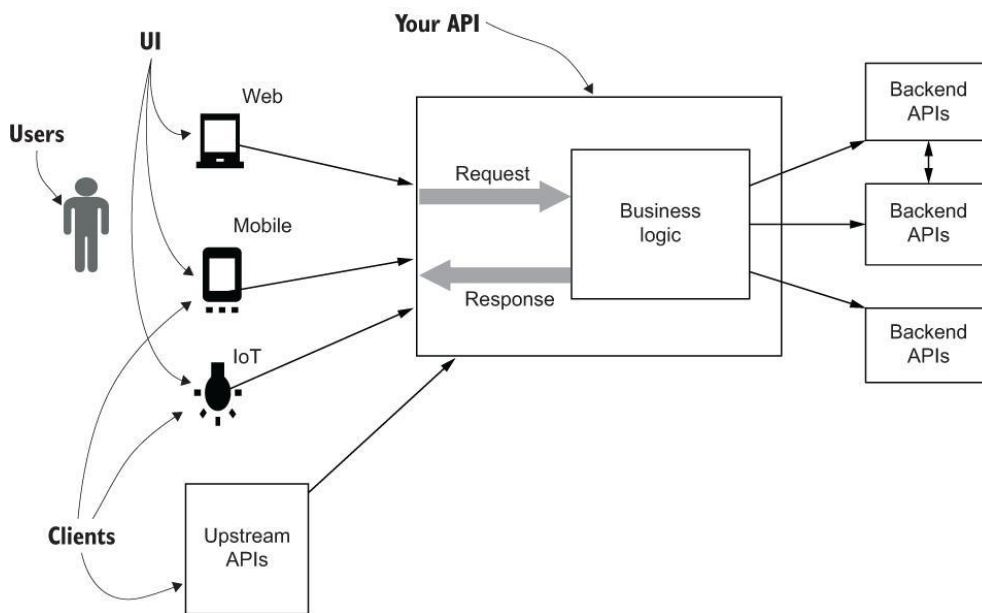


Figure 7. Visualization of both sides of the API (Madden 2020, chapter 1.2)

3.2 Clients

As seen on Figure 7, there is always some kind of a client which works with the API and typically there are many. A client could be any kind of application that is configured to communicate with the API. This client application could again have its own API, thus becoming a part of the “chain” of APIs and have its own clients. A common example of a client application which could utilize an API is a modern-day website (web application), which a user would access through the web browser of a computer or a mobile device. Many kinds of applications that utilize APIs could also be installed on a mobile device from, for example, the application stores provided by Apple or Google. Other examples of physical devices which typically contain applications interacting with APIs are IoT devices, such as smart watches, robot vacuums or even smart home lighting appliances.

A client should not be confused with a user. When referring to users, typically it translates to the people who are working with the client application through some form of a UI. As the user only sees and works with the client application, it therefore acts as another layer of abstraction, this time between the user and the API. The user only needs to know how to use the client application itself, while the application is working with the API in the background, unknown to the user. The client application also usually performs further parsing or formatting of the API’s responses, before deciding what sort of information should be displayed to the user based on them.

3.3 Behavior

The responses returned by an API to the client application could contain data that the user of the client is requesting to see, in various formats. It could also simply contain a message letting the client know of a successful operation if the request was to modify data through a CRUD API, for example. The response could also be blank, or contain error messages, if the client sends requests in invalid format, which the API does not understand. This kind of situation could be caused by a human error with data entry from the user's side, or a bug in the client application. In the case that the client or the authenticated user of the client is not authorized to perform the requested actions, there could also be no response at all. This is a security measure to keep an API from leaking information of how it functions to potentially malicious parties. All this behavior is API-specific and ultimately depends on the design choices made and practices followed during the technical specification of the API. (Harguindeguy 2021.)

4 REST

4.1 Introduction

REST is one of the most, if not *the* most popular way to implement APIs. However, it is not a protocol. A RESTful web API is one that conforms to the constraints of the REST architectural style, commonly referred to as the "5 principles of REST", such as stateless communication and cacheable data (Au-Yeung & Donovan 2020). As that is all REST is, an architectural style, there is no official standard for implementing RESTful APIs. There is however nothing stopping API specifications from making use of what other relevant standards/protocols have to offer, such as HTTP, IRI (URI), JSON and XML. When it comes to REST, utilizing such global conventions is not necessary but doing so can help the API meet the requirements of REST, as seen on chapter 4.2.

Many kinds of APIs that utilize HTTP are often mistaken for REST APIs. This has become a common issue, as described by Pratt (2021) and has resulted in publications directed towards educating people on what REST is and what it isn't, to stop developers calling their APIs something they are not (see e.g., Jain 2019; Pratt 2021). The main selling point of these publications seems to be that simply utilizing HTTP doesn't mean that an API follows any of the constraints of the REST style and thus, using the term "REST" to generally refer to these APIs should be stopped to avoid spreading the misconception. Interestingly, although HTTP is most often a part of REST APIs as pointed out

by Au-Yeung & Donovan (2020), it is not a requirement for creating a RESTful API according to the originator of REST, Roy Fielding (2000, 5.3.2).

Resources

Perhaps the most essential part to understand about REST is the definition of a *resource*. A resource itself can be a reference to any set of static or dynamic information. It could be a reference to a set of existing data entities within the server, or to a concept of something that doesn't exist yet. It could even be a collection of other resources. There are many things that a resource could be, but the main requirement is that it needs to be able to have a name which all related information within the system can be uniquely associated with. (Fielding 2000, 5.2.1.1.)

To better illustrate what a resource is, a "person" could be considered. A person-resource would be a general reference to the *concept* of person, which means that all information on the server related to any single person would be associated with that resource (ibid.). Most likely the server would in this case contain information of several people and with some additional clarification, such as an ID number, the details of just one specific person or a subset of people could be fetched or modified, all by accessing this generic "Person" resource.

As previously mentioned, a resource can also be a concept of something that doesn't exist. For example, if the resource is called "Billionaire" and there exists no entity in the system which can be classified as a billionaire, the resource can still be accessed, although there would be no actual data entities available through it. The billionaire-resource might also return different data at different times since wealth is something that changes over time. If a resource was called "Warmest City", referring to the city which currently has the highest temperature in the world, the information would always point to a single city, but the exact city which the resource is referring to would also change depending on when the resource is accessed.

The reason this is relevant to REST, is because when a RESTful API is implemented, there must be resources specified on the server. When a client is sending requests to the API, each request must be directed to a specific resource. A common way to achieve this is to map a single URL on the web server to each resource, after which the methods of HTTP, such as GET, can be used to interact with any of the server resources.

4.2 The 5 principles of REST

As previously mentioned, the characteristics which allow an API to be called RESTful are often not very clear among developers. This chapter will focus on explaining the 5 primary constraints, *all* of which an API would need to comply with to be called RESTful.

4.2.1 Client-server architecture

RESTful APIs divide the concerns of UI and data storage, delegating them separately to the client and the server (Figure 8). While this might be common behavior for many kinds of APIs, the client-server constraint makes it clear that this is the only correct architectural style to work with, when developing a REST API. (Fielding 2000, 5.1.2.)

Client-server style improves the portability of the user interface by delegating the entirety of user experience to the individual clients' developers, allowing them to not only develop on multiple platforms, but also to consider the needs of the specific users of that application, allowing the RESTful interface to remain completely invisible to the end user. It also makes the API more scalable by removing unnecessary complexity on the server's side. Most importantly, it also allows both the API and the client to evolve independent of each other. (ibid.)

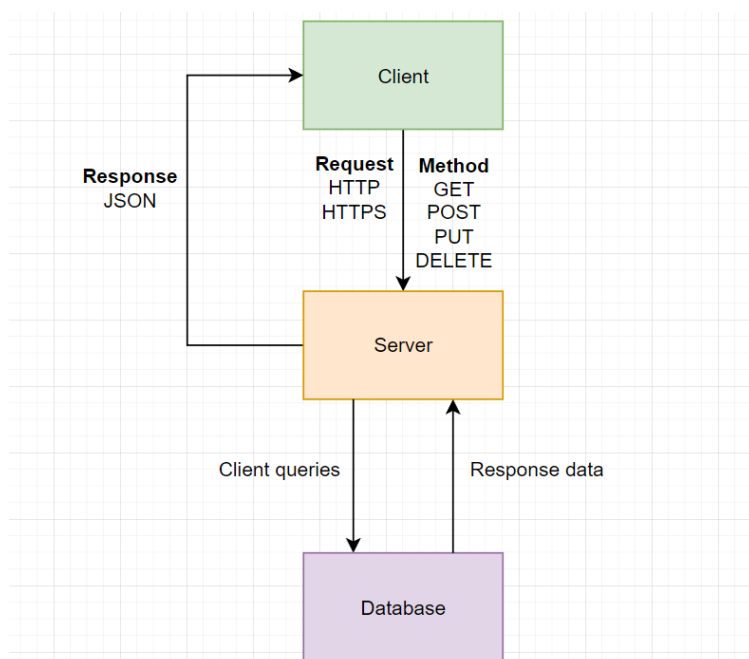


Figure 8. Example of a common REST architecture diagram

4.2.2 Statelessness

Another constraint of REST is statelessness, which defines a requirement for all communication between clients and the server to be stateless. This means that every request sent by clients must contain every piece of information that is necessary for the server to understand the state of the requestor, without utilizing any existing context stored on the server. In other words, every request should be processed in complete isolation, interpreted and processed solely by utilizing information sent along with the request. (Thelin 2021.)

The stateless architecture can be helpful as it makes the API much less complex since no server-sided states are required, increasing reliability through simplicity. Scalability is increased even further, as the same API can be deployed to multiple servers when no states exist outside of individual requests. As both the server and the clients around a stateless API can be very easily changed, the lifespan of the API is also increased. Statelessness can still be limiting, mainly because the payload size and thus network traffic quickly grow for complex queries due to all attributes related to the client's state being included in each request. (Thelin 2021; Gupta 2021.)

4.2.3 Uniform interface

When communicating with an API, generally the client must know exactly how to form the request, and where to direct it to. This kind of information is typically defined in detail in the API specification, which should act as a contract on how the clients are expected to communicate with the API, and how the API is expected to respond. When every component of the system follows the same contract, it results in *uniform* communication.

One of the primary characteristics of a REST service is the uniformness of the API connecting components. According to Fielding (2000, 5.1.5), this is in fact the “central feature” of REST and thus the best way to distinguish it from other network-based styles. Among other things, it means that information is transferred in a standardized way, regardless of which client is requesting said information, the technical details of their implementation of the API client, or the services they further provide. (Fielding 2000, chapter 5.)

This constraint makes it possible for clients to trust that the same exact identifier, a URI for example, can be used to access the same resource every single time. The client must also be able to trust that, although the API might evolve, the data format of the responses provided by the API is not going to change. All resources on the server should be accessible through a common method, an example of such a way being the HTTP GET method. Each resource should also only be identified by exactly one unique identifier (Jain 2019).

HATEOAS is another part of this constraint, and its goal is to decouple the client and the server. According to the restrictions of HATEOAS, a RESTful API should include within each response links that point directly to all related data and any possible interactions with them, regarding the current state of the application. Practically this means that the client needs no prior knowledge of what the current state of the application is, as all the relevant information is contained in the response. It allows for a very dynamic way to develop the REST client, as the clients can dynamically further interact with the server application simply based on the responses. According to Reiser (2018), in an ideal scenario the client can interact with every aspect of the interface simply by following links in the responses, thus being “guided” through the interface by HATEOAS. This is also an effective way of allowing functionality of the server to evolve without breaking existing integrations. It should be noted that although HATEOAS might sound practical in theory, it has also been criticized and even called “useless”. (Reiser 2018.)

4.2.4 Caching

Another constraint of REST is the requirement for the server to make it known with each response whether that data is cacheable or not (Fielding 2000, 5.1.4). Caching means that the client can store the response locally, which in turn means that if the same request is repeated in the future, the client may use the previously stored response instead of sending an actual request to the server (ibid.). This effectively allows full or partial removal of some of the interactions between the client and the server (ibid.). Fielding (2000, 5.3.3) describes the usefulness of caching in a clever way: “An interesting observation is that the most efficient network request is one that doesn't use the network”.

As caching is clearly a big performance advantage, it also strongly supports the choice of HTTP as the application layer protocol with RESTful APIs (Figure 9). This is because caching is a very large

part of the HTTP specification itself, meaning that most clients, for example web browsers, would already be able to take advantage of an APIs caching functionality out of the box. GET requests for example can be cached, kept in browser history and bookmarked. (Gilling 2021; Bush 2020; Fielding 2000, 5.3.3.)

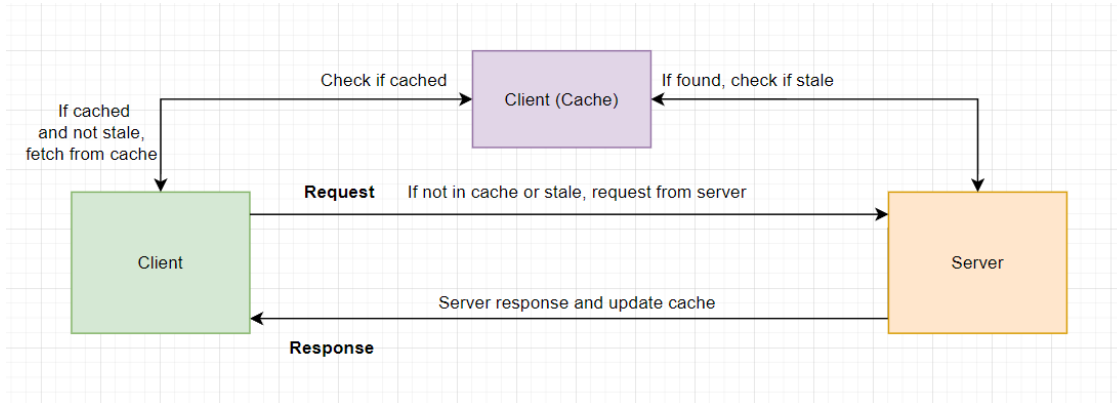


Figure 9. HTTP caching

4.2.5 Layered architecture

The idea behind the layered architecture constraint is to allow the system architecture to be built in a layered manner. In such systems, multiple hierarchical layers of components exist, each of which cannot see beyond the layer they are immediately interacting with. This also results in the same logic being applied to clients, which means that the client might not be able to tell whether they are connected to an intermediary component or the actual end server. (Fielding 2000, 5.1.6.)

Layered architecture reduces the overall complexity of components of the system by promoting their simplicity through independence. As the visibility to layers beyond is restricted, new functionality may need to be wrapped into an additional, independent intermediary component. Examples of such components are load-balancers and proxies, which could be used to increase system scalability and authentication security. (Thelin 2021; Fielding 2000, 5.1.6.)

4.3 Versioning

No clear rules for versioning exist for REST, which means that each application has the responsibility to come up with their own versioning approach, although versioning is not mandatory at all (Bush 2020). As a matter of fact, Roy Fielding, the originator of REST, generally recommends

against the versioning of APIs and believes that the right way to approach change is through API evolution instead (Doerrfeld 2017).

5 GraphQL

5.1 Introduction

GraphQL is a typed query language, purpose of which is to make APIs fast, flexible and developer friendly. According to Byron (2015), the journey of GraphQL began in 2012, when Facebook started rebuilding their native mobile applications. GraphQL was developed as a solution to the problems arising from having to deliver the Facebook news feed to mobile applications as HTML (ibid.). There was a need for a new solution working based on API data instead, along with additional “frustration” regarding the amount of code required to first prepare the data on the server side, and then to properly parse it again on the client’s side (ibid.). These issues resulted in a project, today known as GraphQL (ibid.). Although the specification itself for GraphQL was originally developed by Facebook, it is now being actively governed by the GraphQL Foundation (Frequently Asked Questions (FAQ) N.d.).

5.2 Schema

Schema is the foundation of a GraphQL application. It contains all relevant information about the data structure, making it easy for clients to understand what kind of requests they can make and what to expect from the results. The schema consists of objects that contain one or more fields, which can be used by the clients to specify precisely what they need, in a single query operation. Fields describe the type in which the data is returned. The return type can be any of the scalar types, an object, an interface, a union or an enum. (Hagen, N. 2018.)

Query type is a mandatory type in every schema, as it acts as an entry point describing what read operations can be executed against the GraphQL API, and in what type the data will be returned in. A very basic schema is visualized on Figure 10, where querying the API would return an array on Animals. It is worth noting that when fields are specified without an exclamation mark at the end, they can contain null values. (GraphQL schema basics N.d.)

```

type Animal { #object type
  name: String # Field with a scalar type of string
  breed: String # Field with a scalar type of string
}

type Query {
  Animals: [Animal]
}

```

Figure 10. Sample schema with object and query types

Mutation type is an optional type in the schema, which has an important role in an API endpoint that allows altering data. Mutations are usually implemented when there is a need to perform insert, modify or delete operations in a database. It's a good practice to return a response containing the most recent data in every mutation operation, to the client, eliminating the need to send another query afterwards to fetch the latest data, as expressed on Figure 11. (Wieruch 2019, 35; GraphQL schema basics N.d.)

```

type Animal { #object type
  name: String # Field with a scalar type of string
  breed: String # Field with a scalar type of string
}

type Query {
  Animals: [Animal]
}

type Mutation {
  addAnimal(name: String, breed: String): Animal # Returns a new animal
}

```

Figure 11. Sample schema with object, query and mutation types

5.3 Resolvers

Resolvers are used to fetch data for queries from backend data sources. A resolver must exist for every field in the schema, otherwise a default resolver will check to see if a parent object has the same field defined, and then try to use that to return a value. A collection of resolvers defined in a single object is called a resolver map (Figure 12), which contains all top-level fields with corresponding data types. (Stuart 2018.)

```
#schema.graphql
type Animal {
  breed: String! #
}

type Building {
  buildingType: String!
}

type Query {
  Animal: String!
  buildingType: String!
}

#resolvers.js
const resolverMap {
  Query: {
    Animal() {
      return "Angora"
    },
    Building() {
      return "Flat"
    }
  }
};
```

Figure 12. Resolvers with hardcoded return values

5.4 Queries

As briefly discussed in chapter 5.2, queries are read operations executed by clients to fetch data from an API. GraphQL utilizes HTTP as the client-server protocol, giving clients the flexibility to use any programming languages, client libraries or command line tools such as curl for this process. The only thing that needs to remain constant across implementations is the proper formatting of the queries according to the GraphQL specification and schema. (Stemmler 2020.)

Queries can be sent via HTTP GET and HTTP POST requests, and the server will always return a response in JSON format, as shown on Figure 13. Every request must contain query information at minimum, but additional GraphQL variables can also be passed, which technically enables a GraphQL server to parametrize and hide complex functionality on the server's side where it can be easily re-utilized by multiple clients. GET requests can be considered a typical method for read only operations as they are faster, easier to use and can be cached, which means that it is possible to implement some caching functionality for GraphQL if using them. However, globally unique identifiers do not generally exist for GraphQL which could be leveraged for building caches and POST is usually the way GraphQL is utilized, meaning that caching might often not be an option. (Jacobs 2004; Requests and Responses N.d)

```
#GraphQL query
query {
  animals {
    breed
  }
}

#HTTP GET request to accomplish the same results
http://api.local/graphql?query={animals{breed}}

#Response in JSON format
{
  "data": {
    "animals": [
      {
        "breed": "American Rabbit"
      },
      {
        "breed": "Golden Retriever"
      },
      {
        "breed": "Harlequin Rabbit"
      }
    ]
  }
}
```

Figure 13. GraphQL query and response in JSON

5.5 Versioning

GraphQL has taken a different approach to versioning by providing the means for ongoing evolution of the schema. Because of this, versioning can, *in theory*, be avoided altogether. Evolution is a continuous process where you can introduce and deprecate fields, but existing attributes should not be modified to avoid breaking changes and maintain backwards compatibility. Deprecated fields can then safely be removed when they are old enough and no longer used. (Wieruch 2019, 6.)

5.6 Authentication and authorization

Authentication is the process of verifying an identity. GraphQL doesn't have a take on the process, so the developer must define how it will be done. Authentication middleware can be implemented

on any part of the chain (Figure 14), but it's generally recommended to handle it on the GraphQL server itself. (Simha 2019.)

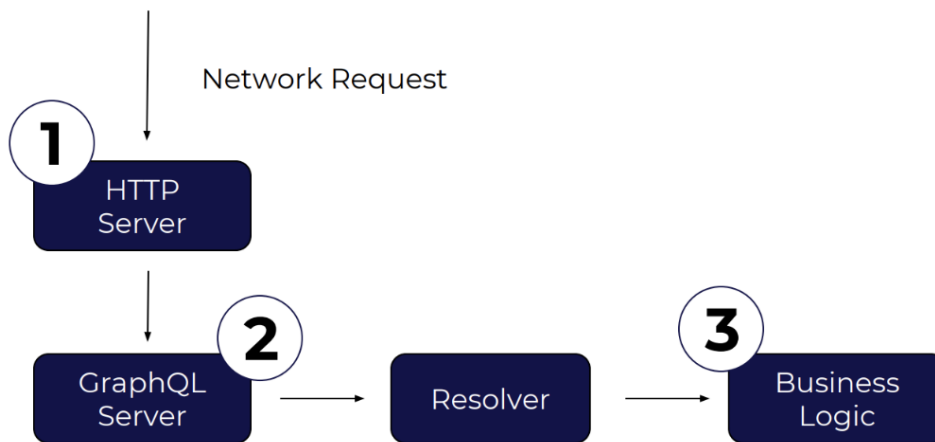


Figure 14. Access control layers (Simha 2019.)

Once the user has logged in, the next step is to verify what the user can see and interact with inside the system. This is called authorization or access control. Even if the API service would be read only, every request should still be authorized to track usage statistics and control what kind of data specific users are allowed to read and work with. (Simha 2019.)

It's a very common scenario to utilize JSON web tokens in the authentication and authorization flow. JWTs have become a very popular choice as they are a simple, secure way to exchange data between parties with very little overhead. After the client has passed the initial authentication phase, the authentication server will send a unique JWT to the client's browser (Figure 15). The client will pass this JWT inside the header field in all further API calls to the server, so it can verify if the user has access to a specific resource. (Get Started with JSON Web Tokens N.d.)

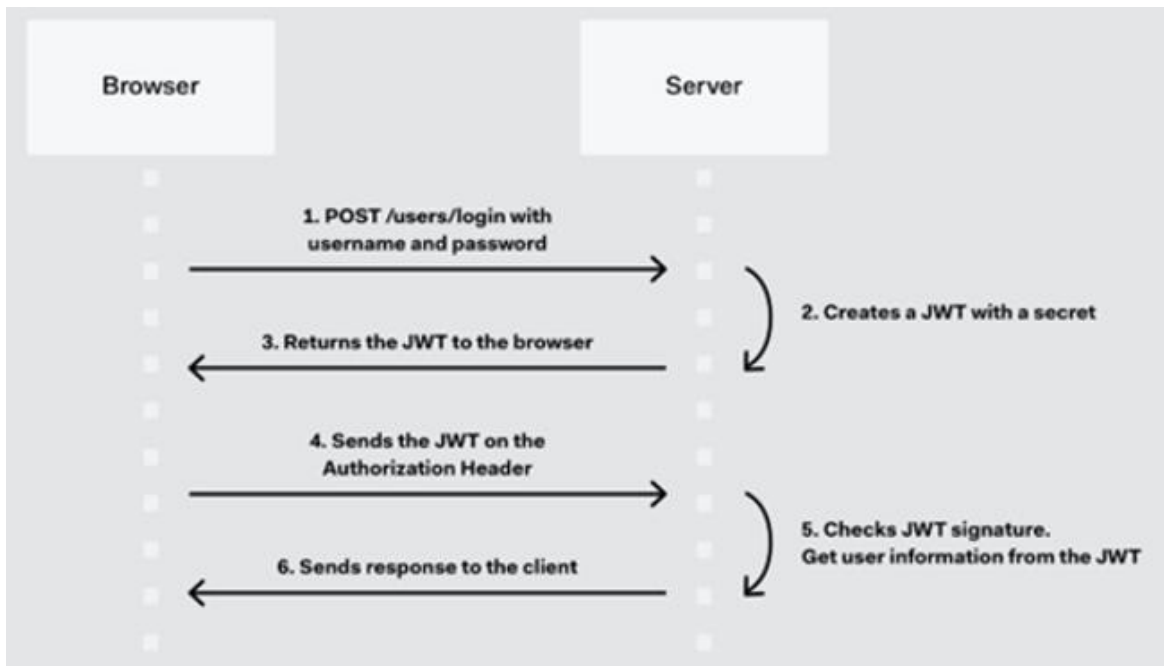


Figure 15. Authentication and authorization flow (Get started with JSON Web Tokens. N.d.)

6 Choosing the right approach

6.1 Unified APIs

Before starting to compare REST and GraphQL, it is worth noting that they do not completely cancel each other out. GraphQL is a robust solution which can also function as a way of bringing multiple existing services into one. This means for example, that if there are many existing REST APIs, one might decide to leave them untouched while implementing a single new GraphQL endpoint. This new endpoint could allow access to all the existing REST endpoints, effectively changing the technology that the clients use from REST into GraphQL as well as combining multiple resources into one, common endpoint. GraphQL makes this possible without requiring any modifications to the existing interfaces, effectively making it possible to keep legacy REST integrations intact while allowing new ones to start implementing GraphQL. (Schultz 2018.)

6.2 Project specifications

While one of the main purposes of an API generally is abstraction itself, in this case the focus further shifted towards abstracting the *deployment and configuration* processes of the API, as de-

scribed in the goals of the research project. This means that it should be as easy as possible to deploy the API regardless of the technical details of the backend application. It also means that the chosen API technology should either already be extremely flexible during the deployment process or allow for a reasonable way to further develop new functionality to improve the existing process. One of the research questions was defined from the perspective of the future users of the API, which must also not be forgotten. GraphQL additionally seems to provide a solution to the initially defined problem of over-fetching, as further explained in the next chapter 6.3.

6.3 Performance

As described by Madden (2020, chapter 1.2.1), GraphQL is mostly concerned with the efficiency of queries and the ability to filter the results in detail. As the client can, to a great extent, add filters to their queries resulting in only specific parts of the data being returned, a RESTful API returning all the data at once might lose when it comes to performing actions on large sets of data, especially if the data is deeply nested and only small parts of it is required. Choosing to use GraphQL could therefore increase performance on either the server's or the client's side, or even decrease the network load. (Oggier 2020, 24-33.)

It should be noted that REST can still outperform GraphQL in some cases, especially if caching is an option for the planned API. This is because GraphQL APIs do not generally support HTTP's caching functionality very well, which RESTful APIs do (Thelin 2021; Bush 2020; Gilling 2021). Some caching options exist for GraphQL as well, but they do not match the full caching capabilities built directly into the HTTP specification, which REST due to its architectural constraints is able to utilize (ibid.). Thus, it is important to notice that there are both pros and cons to GraphQL (Gilling 2021). Investigating its suitability for the application at hand before deciding to implement a GraphQL-based solution over REST is consequently recommended (ibid.).

6.4 Complex data models

One of the main benefits of GraphQL is the way the client is given control over what data should be returned by queries. As seen from the standards and the Converis data model described in chapter 2.2, applications are often built on complex relational data models. The society's needs to have all possible data along with other data related to it readily available are also increasing, as

described by Webb (2015), so being able to retrieve all linked or nested data in just one query might be an ideal scenario for many modern-day applications. Walkowski (2020) also indirectly supports this “data demand” claim by stating that consumers love applications which are innovative, interactive, and provide many services in the same place.

GraphQL accomplishes this by allowing the client to, with a single request, retrieve entities either side by side or in nested relationships (Wieruch 2019, 9). This could reduce the number of requests going through the network in comparison to traditional REST, by shifting the perspective to the client and thus allowing it to combine all its data needs into one clean request (ibid.). A RESTful API on the other hand might require the client to send out not only multiple requests, but also send them into completely different server resources, depending on the data needs. REST can therefore quickly become unmanageable for complex requests (Thelin 2021; Bush 2020).

6.5 Users

The starting assumption is that while giving the client-side application more freedom in shaping the results they wish to receive with a single query, an apparent downside of the GraphQL approach is the steep learning curve. This would result in additional work required from the client’s side (as seen on Figure 16) to build proper queries compared to, for example, a simple GET request to a traditional RESTful API. Interestingly however, according to Brito & Valente (2020, VIII), GraphQL is easier for developers to use than REST, not only for new developers but also the ones already familiar with REST, requiring less effort on many aspects. This challenges the common idea of GraphQL being more difficult than REST.



	 GraphQL	 REST
Architecture	client-driven	server-driven
Organized in terms of	schema & type system	endpoints
Operations	Query Mutation Subscription	Create, Read, Update, Delete
Data fetching	specific data with a single API call	fixed data with multiple API calls
Community	growing	large
Performance	fast	multiple network calls take up more time
Development speed	rapid	slower
Learning curve	difficult	moderate
Self-documenting	✓	—
File uploading	—	✓
Web caching	(via libraries built on top)	✓
Stability	less error prone, automatic validation and type checking	better choice for complex queries
Use cases	multiple microservices, mobile apps	simple apps, resource-driven apps

Figure 16. Comparison of GraphQL and REST (Sikandar N.d.). It should be noted that the information about file uploading is outdated, as it has recently also become a part of GraphQL.

The idea behind GraphQL's difficulty might have to do with the fact that it has not been around for long and that it is only a language. This means that GraphQL on its own is not enough to have a functioning API, but the server and clients need to have their own implementations to utilize the language. This might seem to require more work than a basic REST API, because REST has been around for a long time and widely known solutions exist, especially around HTTP which has been around for even longer and can be directly utilized to implement RESTful APIs. As stated by Gilling (2021), "The beauty of REST is that a developer working with someone else's API doesn't need any special initialization or libraries". GraphQL, although less known than REST, can still be implemented in a very easy manner by utilizing existing applications. One of such applications for the server's side discovered during this research is Hasura, which is further researched in Chapter 7.

6.6 Conclusion

GraphQL contains a variety of advanced features that are better suited for complex APIs, which can benefit from offering a lot of flexibility to the clients. This considers APIs that aim to allow a

large variety of possible interactions to clients, for example when hoping to make future integrations possible ahead of time without additional server-sided work required later, by shifting some of the developer responsibility to the clients. This greatly increases development speed. GraphQL is also useful when wishing to give the clients access to advanced features such as subscription to future data changes. It's also the winning choice if over-fetching, under-fetching or network traffic might become issues with REST, as GraphQL addresses them all.

REST generally works well with simple, resource-driven apps. As concluded by Gilling (2021), REST might be the better choice for APIs which are only concerned with very few entities and simple relationships. If caching of the data is an option, REST might also beat GraphQL in performance, a benefit which becomes less significant as the complexity of the data model increases, since a larger number of queries to various resources is often needed. GraphQL on the other hand is a strong choice for large applications where the API needs to connect with many different types of entities and possibly deep relationships between them. This is because many of its main features, such as simplifying the interaction with complex data models, can then be put to good use.

For the project at hand, it seems justifiable that GraphQL is the right choice for implementing the API. Converis' data model is a complex one, and one of the ideas behind the project was to allow other parties to use as much of Converis' data as possible in their own ways, also considering possible future expansion. Additionally, over-fetching was one of the initially defined issues of the current API, so a GraphQL-based solution would solve that as well.

7 Hasura

7.1 Introduction

When serving a GraphQL API to clients, a GraphQL server is always required. Hasura GraphQL Engine is an open-source GraphQL server which is used to generate instant, real-time GraphQL APIs. Hasura is marketed for its ability to work on any applications utilizing PostgreSQL databases, but there are various other databases also supported such as MS SQL Server, Timescale, Yugabyte, BigQuery and MySQL (beta) with more coming soon, such as Oracle and mongoDB. (Instant GraphQL and REST for Databases N.d.)

Hasura is filled with features designed to make GraphQL APIs perform better as well as make them easier to deploy and develop. This includes the compiling of any kind of GraphQL query into one single SQL query, as well as optimization of the database calls. Hasura not only enables data federation between databases, but also GraphQL and REST services, allowing to bring multiple schemas and existing APIs into one new endpoint, as briefly explained in chapter 6.1. This makes Hasura a great tool for modernizing the use of legacy data. These are only examples of the various features included in Hasura, which will be further described in the next chapter. (A Comprehensive Guide to GraphQL with Hasura, N.d.)

7.2 Features

7.2.1 Automation

Hasura automates many repetitive and time-consuming tasks by design. After a new database is connected, the developer must mark the desired tables and views as tracked before they can be utilized through the API. Hasura GraphQL engine will then generate a GraphQL schema, queries, mutations, subscriptions and relationships between them automatically. No resolvers are created, as the engine itself generates SQL queries from the GraphQL queries, simplifying the schema structure and making the API operations much more efficient. (How Hasura GraphQL engine works, N.d.)

7.2.2 Query performance analysis

As many concurrent users and applications commonly use an API, it's important to ensure that it will perform up to expectations. Hasura takes advantage of the explain statement in PostgreSQL for example, which greatly improves troubleshooting of the performance of SQL queries (Figure 17). Running a query analysis will return a response in JSON format containing total cost, planning time, and execution time amongst other metrics, which might help a developer to understand why a query is taking a specific amount of time, and how it could be further optimized. (Devarkonda 2018; Analyzing Query Plans, N.d.)

Generated SQL

```

SELECT
  coalesce(json_agg("root"), '[]') AS "root"
FROM
  (
    SELECT
      row_to_json(
        (
          SELECT
            "_1_e"
          FROM
            (
              SELECT
                "_0_root.base"."id" AS "id",
                "_0_root.base"."name" AS "name"
              ) AS "_1_e"
            )
          ) AS "root"
    FROM
      (
        SELECT
          *
        FROM

```

Execution Plan

```

Aggregate (cost=16.75..16.76 rows=1 width=32)
-> Seq Scan on users (cost=0.00..12.70 rows=270 width=48)
SubPlan 1
-> Result (cost=0.00..0.01 rows=1 width=32)

```

Figure 17. Generated SQL and execution plan on Hasura web console (Analyzing Query Plans, N.d.)

7.2.3 Authentication

Authentication is not handled within Hasura itself, but instead requires another authentication server to be set up. There are two modes that can be used for authenticating users with Hasura. The first option requires an outside webhook, which is exposed to Hasura. Hasura will then authenticate every request sent to it based on the request's authentication headers, by passing them to the webhook, which will return a list of roles belonging to the requestor. The other option utilizes JWT and requires the client to first get the token from another service, after which it can be used to send requests to Hasura. Hasura will be able to decode and verify the token, after which the user is authenticated. (Authentication & Authorization. N.d.)

7.2.4 Authorization

Whichever authentication method is used, authentication itself is not enough. Hasura will need to, based on the metadata of the request, evaluate access control configuration to decide whether

the requested actions can be allowed to be executed for the requestor. Access control within Hasura is performed based on rules, which can be easily created through the configuration interface. Each rule is linked to a role and can be configured to only consider specific SQL operations, tables or even only specified rows and columns. Example of rule creation through the Hasura console can be seen on Figure 18. (Authentication & Authorization. N.d.)

author **Permission rule granularity: 1) table**

Browse Rows Insert Row Modify Relationships **Permissions**

Permissions

✓ : full access ✗ : no access ⚑ : partial access

Role	insert	select	update	delete	
admin	✓	✓	✓	✓	
user	⚑	⚑	✗	✗	<input type="checkbox"/>
Enter new role	✗	✗	✗	✗	

Role: user Action: insert **Permission rule granularity: 2) role 3) action**

- > Row insert permissions ⓘ - with custom check
- > Column insert permissions ⓘ - no columns
- > Column presets ⓘ - no presets
- > Backend only ⓘ (Know more) - disabled

Save Permissions Delete Permissions

> Clone permissions ⓘ

Figure 18. Access control rule creation in Hasura (Authentication & Authorization. N.d.)

These rules are utilized by utilizing dynamic session variables from the external authentication service during every request. This is done either by having Hasura pass the request headers to the authentication service at every request to receive the session variables, or by decoding them from an already-supplied JWT, depending on the chosen authentication method as previously explained. The session variables contain data such as the default role and list of all allowed roles for the requestor. If there are many roles available, one can be indicated in the request by the client to differ from the default value. (ibid.)

7.2.5 Cloud

As the GraphQL server itself is open-source software, the product Hasura really seems to be selling to their customers is the cloud implementation of it, appropriately named Hasura Cloud. Getting started with the cloud version appears to be easier, as almost everything is set up for the user automatically. There are also additional features that come along with the cloud plan, besides the general benefits of cloud computing. This kind of approach naturally comes with a monthly fee, whereas the open-source version of Hasura can be manually installed as an on-premises solution, which means it is completely free to use. While the cloud option requires no hardware to set up, the open-source version is built to run in Docker containers, which allows it to be set up in almost any kind of environment. (Hasura Products: Cloud, Open Source & Enterprise (On-Prem). N.d.)

7.3 Suitability for the project

GraphQL was already chosen as the API technology for the project, meaning that a server that further eases the use of GraphQL seemed to benefit everyone. As the project was focused on abstraction of the API, Hasura seemed to be a very strong addition to the stack. Hasura is marketed for its ability to be directly connected to data sources including most database systems and other interfaces, even connecting to multiple ones at once. This made it seem like Hasura isn't designed for any specific applications, but instead for being able to bring together as many of different kinds of data sources as possible, unifying them behind a single API.

Hasura also seems to provide answers to problems of the existing Converis API which were highlighted during the project specification. One of the primary issues was over-fetching, to which the GraphQL language itself is a solution, as clients can specify precisely what information they need. Another major problem was the lack of roles, which Hasura solves by allowing *extremely* specific permissions management for different custom-defined roles, even down to the row and column level. This also means that the principle of least privilege can be followed in detail.

One additional note was made to focus on the reduction of developer workload. While GraphQL generally allows shifting some of the work of the API developers to clients instead, it was also noted that reducing the work of the client developers would also be a good thing. As GraphQL had been decided to be the best way to move forward, it didn't end up looking particularly simple

from the clients' perspective. However, as usefulness of the previously mentioned features of Hasura continued to outweigh this seemingly inconvenient matter, it remained to be determined in practice what Hasura could do, if anything, to ease the work of the clients.

Additionally, it is worth noting that there are several benefits and downsides to both on-premises and cloud solutions in general. Cloud computing, however, is a large topic of its own and thus the comparison between on-premises and cloud is far beyond the scope of this thesis. If deciding to implement Hasura, the value of the additional features offered by Hasura Cloud as well as the differences between cloud and on-premises solutions are recommended to be inspected individually. The on-premises version of Hasura was chosen to be implemented for the purposes of this thesis simply due to being the free, open-source option.

8 Implementation

8.1 Prerequisites

Chapter 8 focuses entirely on the practical implementation process of the final product, based on the findings of the previous chapters. It also functions as a detailed set of instructions, based on which a similar solution can be implemented and used with another application, without previous knowledge of GraphQL and Hasura. Detailed information such as specific commands are included to make the implementation reproducible. General technical knowledge regarding installations and configurations performed on Linux-based systems is required to re-implement the product of this project.

RHEL 8

As the virtual server used for the implementation of the API was running a RHEL 8 installation, some changes to the setup process of Hasura were required. The main difference caused by this OS version was the absence of Docker, which was replaced in RHEL 8 by Podman. While the aim of Podman is to completely replace Docker and thus work with existing Docker-dependent software, Podman was still under development and Hasura contained no official installation support for it. Some issues came up during the installation process for these reasons, and workarounds for them are included in this chapter, applying to the specific software versions used at the time. For OS versions other than RHEL 8, or different versions of the software used, the instructions of this

chapter will likely need to be applied in a different way for them to result in a fully functional implementation of Hasura, although the use of Docker would most likely make the process easier.

Web server

There existed a web server and firewall services related to the virtual server used in this implementation, which were both externally managed by the employer organization. What it practically meant during this implementation was that to expose the functionality of Hasura to the network, additional configuration of the existing web service had to be done to let it act as a reverse proxy. Configuration was thus done to forward traffic from the existing exposed web server to the locally installed Hasura service.

This process may highly differ depending on the existing system and other requirements. In this case, the existing web server used to achieve this kind of forwarding was Apache. However, another service could be used instead to achieve the same functionality. Hasura also sets up its own web server which could optionally be exposed directly, by configuring the firewall and thus requiring no proxy at all. The installation or configuration of Apache is not included in this thesis, because it is in no way a requirement for installing Hasura. The firewall configuration is also only mentioned on a generic level, as the detailed configurations will differ depending on the system.

Packages

Some packages are required to be installed before Hasura may be set up. The packages that were pre-installed on the server are `wget` (1.19.5), `python3` (3.6.8) and `python3-virtualenv` (15.1.0-21). These may be installed through any package manager, which in this case was `dnf`.

8.2 Hasura installation

The first step was to get root privileges by entering the “`sudo su -`” command. This meant that “`sudo`” could be omitted from all future commands, as the active user was changed to root. The setup process then began with the installation of Podman, which was done through `dnf`:

```
dnf module enable -y container-tools:rhel8 && dnf module install -y container-tools:rhel8
```

While the installation documentation of Hasura only contains instructions for Docker, and advises downloading it through a custom repository, RHEL 8 has taken many steps to shift away from it in favor of Podman. Docker is quite old and hasn't changed much since release, making it difficult to design modern solutions when the engine itself is starting to lack behind. Podman was designed to overcome these obstacles, so this installation was completed with the natively supported Podman container engine. Version 3.3.1 of Podman was installed.

After enabling Podman, the project directory `/usr/local/crisapi` was created. Within this directory, two additional sub-directories, `hasura` and `venv`, were also created. The following command chain was used to achieve this:

```
mkdir /usr/local/crisapi/hasura -p && cd /usr/local/crisapi && virtualenv venv
```

The directory name “venv” is short for virtual environment. The purpose of virtual environments is to keep individual Python environments as isolated from the system as possible, to avoid possible conflicts when implementing several separate environments within the same system. Next, the created virtual environment was activated as follows:

```
source venv/bin/activate
```

After activating the virtual environment, all changes to be made to the Python environment would be isolated to exist only within the single directory “venv”. A virtual environment such as this one can be exited any time by using the simple command `deactivate`.

At this point, a configuration file was downloaded according to instructions provided in the Hasura documentation. This was placed into the previously created `hasura` directory, by entering the following commands:

```
cd hasura && wget https://raw.githubusercontent.com/hasura/graphql-engine/stable/install-manifests/docker-compose/docker-compose.yaml
```

At this point, a suggested way to continue was to use a tool called *docker-compose*, which is used for automatically setting up and configuring a set of docker containers, based on a single configurable YAML-file which can be executed by this program. This was the file previously downloaded by *wget*, called “*docker-compose.yaml*”.

Podman had been described being able to replace Docker so well that a *docker-alias*, referring to podman, was a possibility to ease the transition. Because of this, an attempt was made to make *docker-compose* compatible with Podman by simply creating the said alias for podman. This was not successful, however, as major differences between Podman and Docker do still exist which resulted in various issues when attempting to execute the compose file.

Another attempted but also initially unsuccessful solution to this problem was to download *podman-compose* instead of *docker-compose*, simply by executing the command “*pip3 install podman-compose*” in the virtual environment. Podman-compose is a project designed to fix the differences and effectively make *docker-compose* files directly compatible with Podman. However, as the project was still in somewhat early development, some issues did exist. Luckily, the issues relevant to this project were fixed by instead downloading what was at the time the latest development branch of *podman-compose*. This was achieved through the command:

```
pip3 install https://github.com/containers/podman-compose/archive/devel.tar.gz
```

The exact version of *podman-compose* installed by this command was 0.1.7dev. Incompatibilities however still existed regarding how differently networking is performed within the “containers” in Docker, and in Podman. These issues resulted in the containers inside the pod not being able to communicate with each other by their names. A change to the compose file configuration was therefore made, changing the references of the “*postgres*” container within the Hasura container to “*localhost*” instead, as seen on Figure 19. This workaround allowed Hasura to properly contact the other container, on which the Postgres database used by Hasura itself for metadata was running. It is worth noting that the default Postgres password should also be changed for security reasons, as well as enabling the `HASURA_GRAPHQL_ADMIN_SECRET`. If choosing to not enable the secret, anyone can access the Hasura web console and make modifications with administrator privileges. Other changes within the compose file may also be made later before moving to production, such as disabling development mode.

```

1  version: "3.6"
2  > services:
3  >   postgres:
4     image: postgres:12
5     restart: always
6     volumes:
7     - db_data:/var/lib/postgresql/data
8     environment:
9     - POSTGRES_PASSWORD: postgrespassword
10 >   graphql-engine:
11     image: hasura/graphql-engine:v2.0.7
12     ports:
13     - "8080:8080"
14     depends_on:
15     - "postgres"
16     restart: always
17     environment:
18     ## postgres database to store Hasura metadata
19     HASURA_GRAPHQL_METADATA_DATABASE_URL: postgres://postgres:postgrespassword@localhost:5432/postgres
20     ## this env var can be used to add the above postgres database to Hasura as a data source. this can be removed/updated based on your needs
21     PG_DATABASE_URL: postgres://postgres:postgrespassword@localhost:5432/postgres
22     ## enable the console served by server
23     HASURA_GRAPHQL_ENABLE_CONSOLE: "true" # set to "false" to disable console
24     ## enable debugging mode. It is recommended to disable this in production
25     HASURA_GRAPHQL_DEV_MODE: "true"
26     HASURA_GRAPHQL_ENABLED_LOG_TYPES: startup, http-log, webhook-log, websocket-log, query-log
27     ## uncomment next line to set an admin secret
28     # HASURA_GRAPHQL_ADMIN_SECRET: myadminsecretkey
29 > volumes:

```

Figure 19. Modified docker-compose file for bringing up Hasura and its internal database. The two environment variables `PG_DATABASE_URL` and `HASURA_GRAPHQL_ADMIN_SECRET` were further modified afterwards.

Finally, to bring up the pod and thus Hasura, allowing the web UI to be accessed, the following command was entered within the hasura directory:

```
podman-compose up -d
```

After the containers were brought up, the Hasura web console became accessible through the port 8080, as seen on Figure 20.

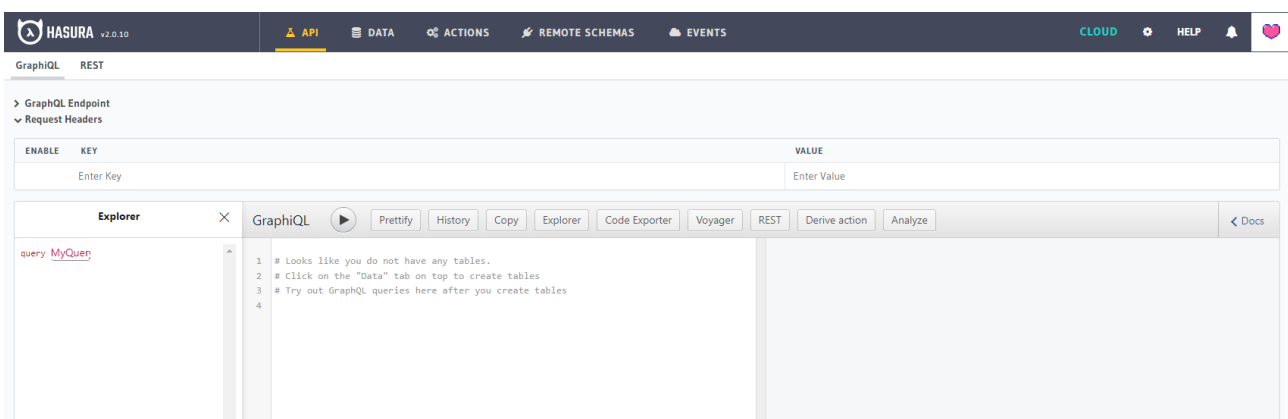


Figure 20. Hasura web console

8.3 Configuration

8.3.1 Data source

Creating a new database connection to give Hasura a source of data was the first step to get started with the configuration process. As seen on Figure 19, the primary data source was specified in the environment variable `PG_DATABASE_URL`, which is the best practice from security perspective, as connection details are kept with Hasura’s metadata. Instead of using the internal Hasura database as a data source as seen on the figure, the connection string in the compose file was further edited to instead point to the database of Converis, existing on another server. The connection string can further utilize environment variables of the host system by using the dollar sign with curly brackets “`{}`”. Here is an example of such connection string, assuming the included variables exist:

```
PG_DATABASE_URL: postgres://${DB_USERNAME}:${DB_PASSWORD}@${DB_HOSTNAME}:5432/${DB}
```

This `PG_DATABASE_URL` environment variable was then added to the configuration of the Hasura container so that the data source could be connected through the “DATA” tab of the web UI, as seen on Figure 21. It should be noted that since the implementation focused on a read-only API, a database user with read-only permissions was used to comply with the principle of least privilege.

The screenshot shows the 'Connect Existing Database' configuration page in the Hasura web UI. It includes the following fields and options:

- Database Display Name:** A text input field containing 'Converis'.
- Data Source Driver:** A dropdown menu set to 'PostgreSQL'.
- Connect Database Via:** Three radio button options: 'Environment Variable' (selected), 'Database URL', and 'Connection Parameters'. A green checkmark and the text 'Environment variable recommended' are shown below these options.
- Environment Variable:** A text input field containing 'PG_DATABASE_URL'.
- Connection Settings:** A blue link with a right-pointing arrow.
- Connect Database:** A yellow button at the bottom right.

Figure 21. Adding a data source to Hasura through an environment variable

Connecting Hasura to an external database usually requires further configuration on the other side to allow the connection. Usually, the firewall of the other server must be configured to let the connection through. Other software, including the database itself, might also block the connection. An example that came up during this implementation was that Postgres itself prevented the connection because the host-based authentication file was used and not configured properly to allow connections from the API server. The tasks to be performed on the DB server differ greatly depending on individual configurations.

8.3.2 Data definitions

When connecting Hasura to the database of an existing application, especially a large one, chances are there are lots of foreign keys representing relationships between different tables in the database. It is also entirely possible that there aren't any relations where there should be, meaning that they would need to be defined to simplify querying. This becomes problematic if the developer of the API doesn't wish to perform permanent modifications to the database.

Whether there are existing relationships in the DB or not, Hasura comes to the rescue by allowing the easy definition of custom relationships between tables through its graphical UI. If the database already contains foreign keys, Hasura will detect them and suggest adding equivalent GraphQL relationships automatically. Otherwise, the tables and their linked columns must be selected manually to create the relationship. What makes this great is that although relationships are created for the external DB, it is done on the API level, meaning that the DB is not modified at all. Another similarly useful feature is the easy renaming of existing data, again on the API level. These two features combined allow the API developer to make tables and their data more readable and thus easier for the API user to understand, without touching the source database. These features are where Hasura shines, especially when it comes to difficult-to-read legacy data, or tables without well-defined relationships.

While the Converis database comes with well-built materialized views that are easy for the eye to read as well as to query, they have the clear downside of having to be re-created regularly. The creation of the materialized views is a somewhat time-consuming process and means that the available data is only as recent as the latest build of the materialized views. This works well for

things such as daily tasks since the views can be easily regenerated, but when it comes to unpredictable API calls, real-time data is almost a must. The problem with this is that Converis' actual data is contained in tables that are neither easy to read nor to query, at the first glance. Some relations do exist, but there are also missing ones. This means that both the data relationship and renaming functionalities of Hasura can be put to good use when enabling access to real-time Converis data.

One of the hoped outcomes of a new improved API that had previously been emphasized by the employer was the easier fetching of publication data from Converis. Based on this, publication data retrieval was the first proof of concept chosen to be implemented to the API. To allow Hasura to retrieve information on Converis publications, the relationships were defined as seen on Figure 22.

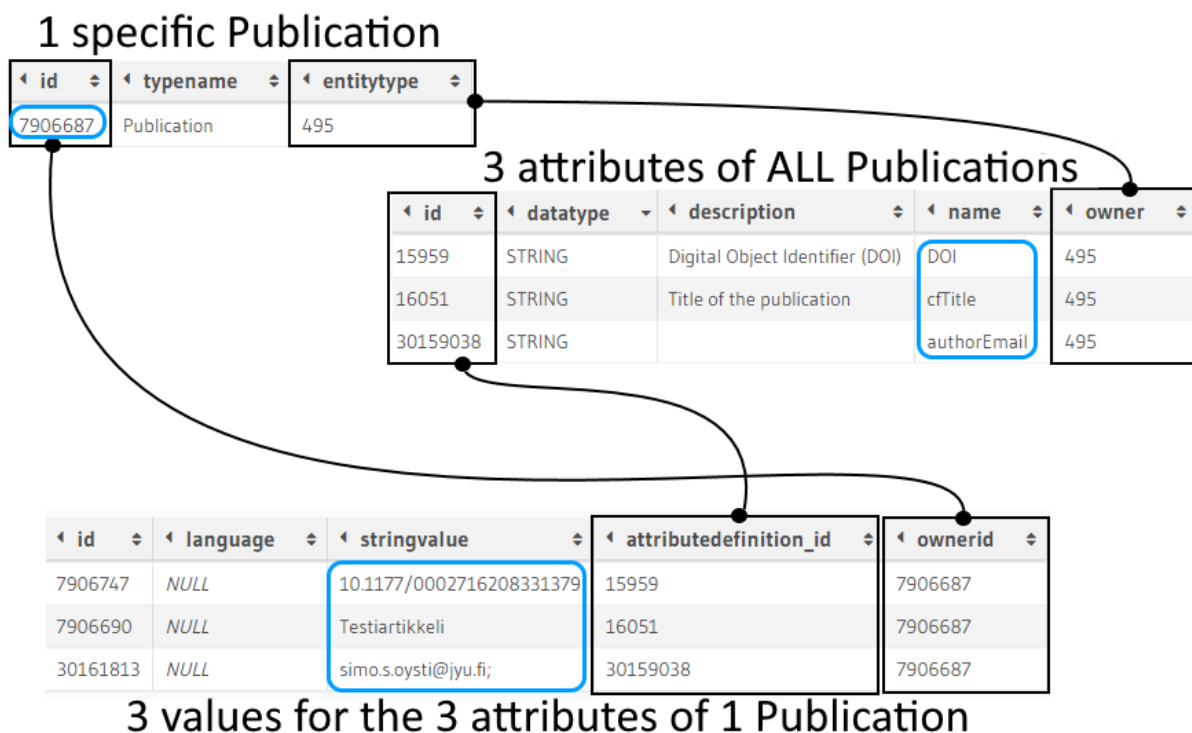


Figure 22. Converis Database relationships. Blue highlights represent the data to be joined together. Black connections represent relationships defined within Hasura, some of which already existed as foreign keys in the DB.

8.3.3 Queries and API endpoints

A basic query

After all the relationships necessary to fetch a publication's attributes in a single query were created, writing the actual query was the next step. With Hasura, this can be accomplished even without any knowledge of the GraphQL language. This is possible through the graphical editor "GraphiQL" which is conveniently integrated into the Hasura web console, as previously seen on Figure 20. A very basic query called "GetPublication" was created, namely designed for retrieving data on a specific publication. A simple condition was written to check for the ID and type in the table of entities, followed by selecting which data to display through the previously defined relationships. The query can be seen on Figure 23.

```

1 query GetPublication{
2   dataentity(
3     where: { typename: { _eq: "Publication" }, _and: { id: { _eq: 7906687 } } }
4   ) {
5     string_attributes { <--- A named relationship, referring to the table which
6       stringvalue          contains all string values.
7       attributedefinition {
8         name
9       }
10    } ^ Choose which values should be fetched:
11  }   - The string value of the attribute
12 }   - The name of the attribute in another table, through a named relation

```

Figure 23. GetPublication v1, querying data based on relations. Returns all string attributes and their values for the publication that has the specified ID.

A twist to the development process was the fact that all attribute values of different datatypes had their own tables. As previously seen on Figure 22, all mentioned values were strings. If hoping to also fetch numbers, dates or binary values for example, more tables and relationships would need to be introduced. To prove the concept of multiple datatypes and fetch some more commonly needed attributes, number values were added to the query along with the strings. This was done simply by copying the existing code block and changing some names to refer to different relationships and columns. The new, updated query which fetched both strings and number values can be seen on Figure 24.


```

1 query GetPublication{
2   dataentity(
3     where: { typename: { _eq: "Publication" }, _and: { id: { _eq: 7906687 } } }
4   ) {
5     string_attributes {
6       stringvalue
7       attributedefinition {
8         name
9       }
10    }
11    number_attributes {
12      numbervalue
13      attributedefinition {
14        name
15      }
16    }
17  }
18 }

```

Figure 24. GetPublication v2, querying values through several relation paths. This version of the query additionally fetches attributes that have number values, which are also in a separate table. The added block of code is highlighted in red.

Now the query started to look like with more datatypes to soon be added, it would become inconvenient and perhaps even too complex of a requirement for the clients to write, especially for a task that assumedly will be repeated frequently and across many clients. Luckily, something that at this point came as a very positive surprise, was that Hasura in fact allows the storing of queries on the server's side, by wrapping them in a simple "REST" endpoint for easy, repeated calls. Before moving on with this approach, it is worth noting that again, this time in practice, the endpoint is by default called REST, although it does not necessarily comply with all the principles of REST. Thus, it will from now on be called a HTTP endpoint.

This approach of wrapping the functionality would make little sense with the ID of a specific publication entity written within the query itself. To make the query much more useful for various purposes, the ID was instead replaced by the variable *\$id*, as seen on Figure 25. This allows the clients to simply supply an ID of their choice with the request, which GraphQL conveniently substitutes in the stored query.

```

1 query GetPublication($id: bigint!){
2   dataentity(
3     where: { typename: { _eq: "Publication" }, _and: { id: { _eq: $id } } }
4   ) {
5     string_attributes {
6       stringvalue
7       attributedefinition {
8         name
9       }
10    }
11    number_attributes {
12      numbervalue
13      attributedefinition {
14        name
15      }
16    }
17  }
18 }

```

Figure 25. GetPublication v3, introducing variables. The previously hardcoded ID has been replaced by a variable, which can be supplied individually by clients and then substituted by GraphQL itself.

At this point the query looked like something that could be repeatedly utilized by multiple clients and for various publications. Wrapping this functionality in a simple HTTP endpoint could therefore be beneficial and it was done by clicking the “REST” button on the GraphiQL interface within Hasura console. After supplying the required information, as seen on Figure 26, the endpoint became immediately accessible.

Create Endpoint

Name
GetPublication

Description
Gets information on the given publication. Requires the ID of the publication and returns all attributes with string or number values.

Location
GetPublication
http:// hostname /api/rest/GetPublication

Methods
 GET
 POST
 PUT
 PATCH
 DELETE

Figure 26. Wrapping a whole GraphQL query within a simple HTTP endpoint

The new HTTP endpoint could be tested by simply sending a POST request to the correct URL and including the required ID parameter in a JSON-formatted body. This request could be made with the simple command line utility *curl*, for example. However, a more graphical approach with an application called Postman was chosen instead. Postman is visually very clean and clear on how everything functions, and it therefore makes HTTP requests very easy, even if the user is not very familiar with them. A new HTTP POST request was created in Postman, and it was directed to the newly created HTTP endpoint. The ID of a specific test publication was included in a JSON body according to GraphQL specification. As seen on Figure 27, fields of the publication were successfully returned.

The screenshot shows a Postman interface for a POST request to `http://[hostname]/api/rest/GetPublication`. The request body is a JSON object:

```

1 {
2   "id": "7906687"
3 }
4

```

The response status is 200 OK. The response body is displayed in a pretty-printed JSON format:

```

1 {
2   "dataentity": [
3     {
4       "string_attributes": [
5         {
6           "stringvalue": "Öysti, S. (2021). Testiartikkeli. In S. Öysti (Ed.), Testiemoartikkeli. https://doi.org/10.1177/8
7           "attributedefinition": {
8             "name": "referencePlain"
9           }
10        },
11        {
12          "stringvalue": "111 Mathematics;112 Statistics and probability;113 Computer and information sciences;"
13          "attributedefinition": {
14            "name": "techAreaNames"
15          }
16        },
17        {
18          "stringvalue": "",
19          "attributedefinition": {
20            "name": "jufoChannelName"
21          }
22        },
23        {
24          "stringvalue": "",
25          "attributedefinition": {
26            "name": "homeoadeURI"

```

Figure 27. The GetPublication HTTP endpoint in action. All fields containing strings or numbers as values were returned by the API.

While this API response might seem completely fine, one of the problems initially outlined was the over-fetching of data. As this query always returns every attribute, it doesn't match the goals of the project. To make this query perform according to the expectations, an additional way to choose the returned fields during each query was required. Like the approach used for supplying the ID through a *big integer* variable, a *list of strings* variable can similarly be created, against which attribute definitions can be checked and filtered. This additional variable and relevant conditionals were added to create the final version of the GetPublication query. The final query and its test results can be seen on Appendix 1.

8.3.4 Authentication

Hasura requires an external auth server to authenticate users, as previously described in chapters 7.2.3 and 7.2.4. Passport.js, a general-purpose authentication middleware, was chosen to be set up to authenticate any calls to the API by using bearer tokens. It functions on the Express Web Framework, which is a very minimal web server built on Node.js.

Firstly, the official Hasura repository was fetched. Inside of it exists the previously mentioned, community managed Node application Passport.js, which was going to be set up to manage HTTP authentication via bearer tokens. The following commands were used to download the repository and navigate to Passport's directory:

```
git clone https://github.com/hasura/graphql-engine  
cd graphql-engine/community/boilerplates/auth-webhooks/passport-js
```

NPM was also installed, which is used to manage Node.js packages. Running the command "npm install" within a package directory retrieves all dependencies and places them in a folder named "node_modules", making it very easy to deploy, manage and isolate Node.js applications. The following commands first download and install npm through the dnf package manager, then install Passport.js and its dependencies, including Express, after which the fresh installation is moved to the crisapi project directory:

```
dnf -y install npm
```

```
npm install
```

```
mv passport-js /usr/local/crisapi/
```

```
cd /usr/local/crisapi/passport-js
```

The web server binds to port 8080 by default, but that cannot be used as it's already reserved by Hasura. Thus, the app.js file had to be modified slightly to change the port binding. The port 3000 was chosen to be used. The file was opened for editing with the command "nano app.js" and the existing line "app.set('port', process.env.PORT || 8080);" was modified so that the port 8080 became 3000 instead.

Another Postgresql database server was installed locally, which is where the authentication credentials were going to be stored by the Node.js application. The command "dnf install postgresql-server -y" was used to perform a fresh installation.

The database server couldn't be started with default configuration, as port 5432 also happened to be reserved by Hasura's database. The port was changed to 5433 instead, by opening the file "/var/lib/pgsql/data/postgresql.conf" and changing the line "#port = 5432" to "port = 5433". SELinux had to be additionally configured to permit this new port, which was done with the command "semanage port -a -t postgresql_port_t -p tcp 5433"

The active user was changed to postgres, so the database configuration files could be created. Trying to do as root will simply cause an error, as postgresql cannot be run as root, and the service account needs access to the configuration data.

```
su - postgres
```

```
initdb -D '/var/lib/pgsql/data'
```

The active user was then exited back to the root account with the command "exit", so that the postgresql service could be started as well as added to the system startup configuration. This was accomplished with the following commands:

```
systemctl start postgresql && systemctl enable postgresql
```

Once again, the postgres user was activated with “su - postgres”, this time for creating the database as well a database user for the authentication service. The following two commands were used:

```
createuser hasura_authuser -p 5433
```

```
createdb hasura_authdb -p 5433
```

At this point, the database was up and running, also being accessible through the new *hasura_authdb* database user. The command “psql -p 5433” was used to locally initiate a connection to the database while logged in as the postgres user. Two commands were to be executed within the database, which would first set an encrypted password for the newly created hasura user, as well as grant full privileges on the new *hasura_authdb* table for that user. These were the SQL commands:

```
alter user hasura_authuser with encrypted password 'hasurasecret';
```

```
grant all privileges on database hasura_authdb to hasura_authuser;
```

The database connection was then terminated by exiting the client through the psql meta command “\q”. At this point, the postgres user was no longer needed so the root user was activated once more. Next, the Knex.js query builder was used to automatically generate the database schema. This was done by first navigating to the directory “/usr/local/crisapi/passport-js” and then running knex with the command “node_modules/.bin/knex migrate:latest”

An environment variable containing a connection string for the new database was then created, so that the node application server could connect to it and store user data. The variable’s name `DATABASE_URL` is hardcoded within the application. It was also appended to the root user’s bash profile, so that it would persist on system reboots. The following command accomplishes both:

```
echo export  
DATABASE_URL=postgres://hasura_authuser:hasurasecret@localhost:5433/hasura_authdb >> ~/.bash_profile
```

An internal environment variable “HASURA_GRAPHQL_AUTH_HOOK: http://localhost:3000/webhook” was then added to Hasura’s pod through the docker-compose.yml configuration file, to specify that Hasura should authenticate any API users by passing their requests to the new authentication service at port 3000. Instead of the term localhost, however, the actual hostname of the local machine was entered into the variable or the Hasura pod was unable to contact the webhook. Afterwards, the Hasura environment was restarted to activate this configuration change, which also required the virtual environment to be activated since that was where podman-compose had been originally installed:

```
cd /usr/local/crisapi/hasura
```

```
source ../venv/bin/activate
```

```
podman-compose down
```

```
podman-compose up -d
```

After once more navigating to the passport-js directory with the command “cd ../passport-js”, the node application was then started with by entering “node app.js”. At this point, the Node application process will attach to the current terminal, so until it has been set up properly as a background service, a new terminal must be opened to continue working while it is running.

As the authentication service behind port 3000 wasn’t opened to be accessible from the outside of the local machine and Hasura, which reduces attack surface from a security perspective, a local POST request was created with curl. This was done to the `http://localhost:3000/signup` route with the purpose of testing the user registration functionality of Passport.js. A new user was successfully created and a bearer token for it assigned, as seen on Figure 28. This bearer token could later be used to authenticate with any Hasura endpoints.

```
[root@~]# curl --header "Content-Type: application/json" \
> --request POST \
> --data '{"username":"user1","password":"secretpass", "confirmPassword":"secretpass"}' \
> http://localhost:3000/signup
{
  "id": 1,
  "username": "user1",
  "token": "7131e97d7f59d93a535e10d64c620d19"
```

Figure 28. Testing new user signup with Passport.js

Passport also comes with a */login* route, which can be used to retrieve the token if it is forgotten but the username and password are known. This on its own has little use, but if reducing the infinite lifespan of tokens and opening the login route to users to allow the generation of temporary session tokens, it does hold potential from a security perspective, which is further explained in chapter 10.3. The login functionality was also tested by sending a request, as seen on Figure 29.

```
[root@localhost ~]# curl --header "Content-Type: application/json" --request POST --data '{"username":"user1","password":"secretpass"}' http://localhost:3000/login
{"id": 1,
 "username": "user1",
 "token": "7131e97d7f59d93a535e10d64c620d19"}
```

Figure 29. Testing Passport.js token retrieval through the */login* route

The default code of Passport.js has been written in a way that all users who successfully authenticate, will receive the role called “user”. This might not be optimal in the long term, but it works for demonstrating the permissions management functionality. It is also very simple to build additional logic for role handling, as the code is simply a JavaScript function that returns a specific role name when requested by Hasura. This “handleResponse” function is further described in chapter 10.3.

8.3.5 Authorization / Access control

The authentication service only takes care of telling Hasura what the role of the user sending the API request is, by checking the authentication header. It falls to Hasura to make sure that this role has access to the correct data. Hasura once again has made it very clear how role permissions can be set up for different database operations and different tables even precisely based on specific columns and row data. An example of granting the previously mentioned “user” role full access to the SELECT statement without any restrictions to specific columns or rows can be seen on Figure 30. This only applies to a single table, so the same must be repeated for all tables relevant to the query, also considering data on both sides of the relationships used. In this case, this permission was applied to the tables “dataentity”, “attributedefinition” and both tables that contain values for string and number attributes.

Permissions

✓ : full access ✗ : no access ▾ : partial access

Role	insert	select	update	delete	
admin	✓	✓	✓	✓	
user	✗	✓	✗	✗	<input type="checkbox"/>
Enter new role	✗	✗	✗	✗	

Close Role: user Action: select

▼ Row select permissions ⓘ - without any checks

Allow role user to select rows:

Without any checks

With custom check: ⓘ

Limit number of rows: ⓘ

▼ Column select permissions ⓘ - all columns

Allow role user to access columns:

id datatype default_expression description editable multilanguage name choicegroup owner logtime

For **relationships**, set permissions for the corresponding tables/views.

Figure 30. Granting the role "user" unrestricted SELECT access to a single table. "Row select permissions" allow defining checks to only allow interaction with rows containing specific data. Similarly, columns can be hidden from the role by de-selecting them.

After the permissions were set, it was possible to perform the final test of executing queries as an authenticated user. Postman was again used, but this time the query should not return any data since the requestor is not yet authenticated. This test can be seen on Figure 31.

POST http://[hostname]/api/rest/GetPublication

Params Authorization ● Headers (8) Body ● Pre-request Script Tests Settings ●

Headers Hide auto-generated headers

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Authorization ⓘ	Bearer invalid_token_test	
<input checked="" type="checkbox"/> Content-Type ⓘ	application/json	

Body Cookies Headers (6) Test Results Status: 401 Unauthorized

Pretty Raw Preview Visualize JSON

```

1
2  "path": "s",
3  "error": "Authentication hook unauthorized this request",
4  "code": "access-denied"
5

```

Figure 31. Unauthorized POST request from an anonymous user

Hasura seemed to correctly reject the request when a valid bearer token was not included in the Authorization header. Similarly, the functionality was tested with the correct token, which can be seen on Figure 32. Hasura accepted the second query because the authentication service returned the role “user” for the provided token, which had been allowed access to the relevant tables.

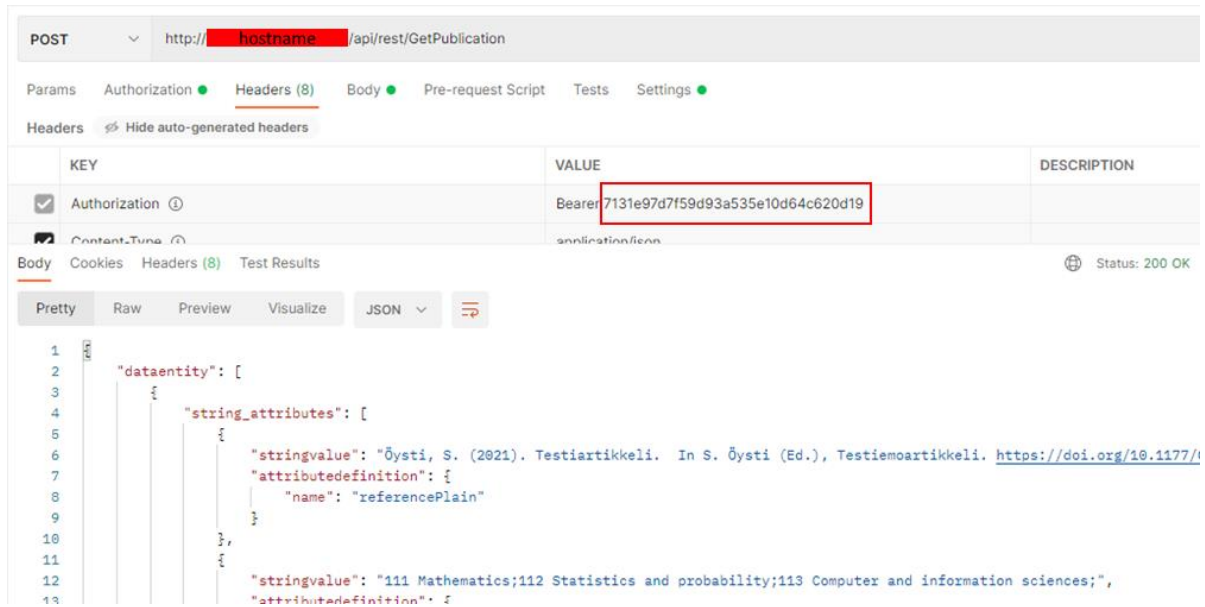


Figure 32. Authorized POST request from an authenticated user. The bearer token previously retrieved from the authentication service is included in the Authorization header.

8.3.6 Logging

Hasura logs everything by default to the Hasura’s podman container logs. The log entry contains role and user information, so that it can be used to identify which client has sent that specific query, also including the variables and SQL syntax. The level of logging can be further adjusted by modifying the `docker-compose.yaml` configuration file, and some logging types can optionally be disabled completely. As seen on Figure 33, Hasura produces a JSON object entry for every query received, containing relevant details such as the role and user-id, as well as the IP address of the caller. Exact SQL queries are additionally logged, which means that every action is traceable.

```

{
  "type": "http-log",
  "timestamp": "2021-12-02T18:51:43.855+0000",
  "level": "info",
  "detail": {
    "operation": {
      "query_execution_time": 6.6906935e-2,
      "user_vars": {
        "x-hasura-role": "admin",
        "x-hasura-user-id": "1"
      },
      "request_id": "68dd4c37-1313-4981-8a1b-4123d1253091",
      "response_size": 1317,
      "request_mode": "non-graphql",
      "request_read_time": 2.93e-6
    },
    "request_id": "68dd4c37-1313-4981-8a1b-4123d1253091",
    "http_info": {
      "status": 200,
      "http_version": "HTTP/1.1",
      "url": "/api/rest/GetPublication",
      "ip": "130. [REDACTED]",
      "method": "POST",
      "content_encoding": "gzip"
    }
  }
}

```

Figure 33. Example entry From Hasura's logs (JSON)

9 Results

All the implemented functionalities were individually tested to be functioning properly during implementation. All of this was documented in chapter 8, leaving no functionality untested. Answering the initial research questions is however important for defining the results of the research, prior to making conclusions. The primary research question and sub-questions were determined in chapter 1.4. The primary question will be answered first, after which answers to the more detailed sub-questions will complete the chapter.

Primary question

The primary research question was at an early stage defined to be *“How to design and implement a read-only API solution well suited for an abstract, existing application?”*. Although it is a very broad question to which there must be many valid answers, one such answer can be concluded based on this research project’s results.

One way to begin the design process is to discover existing tools that hold potential regarding the project goals, which would then be researched in detail. Relevant research on how applications are typically built and whether some standards or common practices have effect on how they are designed, should also be carried out. Through these findings, it is possible to conclude what kinds of related tools and technologies currently exist and which ones are commonly used at the present. This enables the further research of whether there are any issues such as common bad practices, misconceptions or other areas of improvement. Ultimately, it is possible to conclude which tools *can* be used to improve on the existing issues. Some comparison must then be made, based on which the right tools can be selected for the implementation phase.

When choosing to develop a read-only API, the choice already supports abstraction more than a typical CRUD API, due to avoiding unwanted side effects from various application-specific events related to data modifications. When reading data, usually authenticating and authorizing the user as well as logging their actions is enough. All this functionality can be implemented on their own separate layer, meaning that ignoring the application-specific details of the source is a real possibility. Read-only API as a concept therefore supports abstraction right from the beginning, although focus must be kept on finding or developing tools that also support abstraction.

How easy is it to deploy the final product or make changes to it?

The installation of Hasura was shown to be a very easy process, although the use of RHEL 8 and Podman proved to be somewhat problematic at first. The part of the implementation that could be considered most far off from a straightforward deployment, was the inclusion of an external authentication service, which is a requirement of Hasura if hoping to restrict the access of any API users. Although Passport.js was chosen to be used for demonstrating authentication with Hasura and it was very easy to set up, it is in no way the only option, nor claimed to be the best one in any form, meaning that there are a lot of different options that the developer must choose from. If a secure authentication service already exists or can be set up beforehand, the rest of Hasura's deployment and configuration is very straightforward. Additionally, if Docker is available to be used instead of Podman, which might be the case for many systems, the deployment process described will become even simpler as the official Hasura instructions will also support the installation.

During the configuration of various Hasura features, it became clear that Hasura manages to turn otherwise complex configurations to very simple ones through its web UI, meaning that making changes during deployment or in the future doesn't require a lot of development time. Adding a data source was accomplished in a matter of minutes, and so was the tracking of tables or adding new relationships between them, ready to be used within queries. Adding new roles within Hasura and managing their permissions was also extremely simple.

Is the final product a considerable API option for other applications?

Directly connecting the database of Converis to Hasura was possible, while no attention to the specifics of the Converis application itself was required during implementation. This is something that holds potential to be very useful for many other applications as well. Combined with how easy the deployment and configuration processes are, Hasura can certainly be recommended as a considerable API option for other applications, with the exclusion of Passport.js, as the best authentication service to use is another subject entirely and should be evaluated based on individual requirements. Additionally, as this thesis focuses on a free open-source implementation of the on-premises Hasura installation, a recommendation cannot be formed on whether the Hasura Cloud version might be a better choice for a specific implementation.

How far could the API be reasonably abstracted?

The answer to this research question of course depends on the definition of "reasonable". If first considering the abstraction of the *deployment process* of the API, one way to understand the question could be to consider whether there is an actual need for further abstraction or not. One might ask for example, are there any potential application developers who might not be able to easily adapt the current API solution for their application, or are there systems which might not be able to utilize setup process like the one described in chapter 8, or use Hasura at all? It is true that the deployment processes might differ to some extent. However, being able to utilize either Docker or Podman forms a highly portable base for the installation, which most systems should be able to utilize, resulting in no need for additional abstraction of the installation.

The requirement of an authentication service on the other hand, being outlined as a more complex part of the implementation process, has no room for further abstraction. This is because the requirement is already extremely abstract. The authentication service is allowed to differ greatly

between implementations. There might even be existing services that developers could utilize with Hasura. For the authentication server to function with Hasura at the most basic level, it only needs to send one specific HTTP response to one specific HTTP request, meaning that it could even be easily hard coded with a programming language of the developer's choice. Therefore, hundreds of potential tools exist for implementing the service to best suit the system at hand.

On the other hand, if considering how abstract the API itself can be, Hasura does support most of the popular database systems on which applications are almost always built. Hasura also allows the developer to bring any number of data sources together, not only limited to actual databases, but also remote APIs. These facts combined mean that at its best, Hasura could abstract most, if not all, existing data sources, be it legacy or modern, into one endpoint for its clients to utilize. Additionally, considering how far Hasura already manages to abstract the configuration processes within its UI, attempting to take those further would be questionable at the least.

Is the amount of work required from the API clients' developers reasonable?

This might be where the API was not expected to excel at all before the implementation started. As a surprise, the exact opposite was proven. Hasura has the option to allow the *API developers* to selectively take care of the creation of complex queries, parametrize them, and then deploy them within a minute to a new HTTP endpoint. From there, multiple users can utilize the functionality with methods they are already familiar with, without any coding or additional learning required. All this is possible, while also keeping the complexity and thus the freedom of writing queries that clients are given along with a GraphQL-based approach over traditional HTTP/REST APIs.

10 Conclusions

10.1 The chosen tools

While the GraphQL language is a great, robust tool for developing APIs, it also comes with its own complexity. Hasura aims to remove the complexity by making the implementation and development of GraphQL APIs instant and easy, and it seems to do so extremely well. Not only does Hasura make it possible to create complex GraphQL schemas and perform queries graphically without writing any code, but it also makes it possible to easily wrap the GraphQL functionality within additional HTTP endpoints by the click of a button. Effectively this keeps the benefits of GraphQL and

Hasura even while allowing the API developers to give clients the freedom of choosing whether to work with the familiar methods of “good old REST” instead of adapting GraphQL.

There wasn't much existing research or other information available on Hasura, besides their own documentation. This is possibly due to Hasura being a very recent extension to the also quite recent GraphQL language, while major technological adaptations such as REST can take years or sometimes even decades to develop. Based on the findings and testing of Hasura, it functions very well in the intended way and is thus a good choice for deploying GraphQL APIs.

Although GraphQL and Hasura ended up being the chosen technologies based on the research carried out in this thesis, it doesn't mean they are the only ways to implement these kinds of APIs. When such solutions are implemented with other available technologies, comparison between their details and the ones of this implementation can be made. Until then, it is difficult to establish clear guidelines on choosing Hasura specifically over options which haven't been included in this thesis. The recommendation of Hasura can only be made based on the usefulness of GraphQL and Hasura, as witnessed through the research findings, experimentation and testing during this project.

10.2 Challenges

Podman, the utility for managing containers and container images is integrated into the RHEL 8 OS. Podman has several advantages over Docker and in theory should be fully Docker compatible, but it can come with issues. Hasura's container installation guidelines have been designed around Docker and attempting to complete the deployment with Podman will result in a failure. For example, networking is handled differently between Docker and Podman, and sometimes it's necessary to switch to an older release or a development branch to overcome a specific issue. This is very likely related to how recent Podman is, and majority of these incompatibility issues will likely be solved in future releases.

As discussed previously, Hasura doesn't come with any kind of kind of authentication by default, and instead requires the use of an external solution. It's up to the developer to decide how authentication will be implemented, which can be problematic if the developer doesn't have much

previous experience with backend authentication services. A handful of community made solutions exist, but many of them are also either too basic, or built for older package versions that come with known security vulnerabilities. Commercial authentication solutions are an option, but they are often proprietary and come with a price tag, which brings its own issues and limitations.

10.3 Future expansion

Additional roles

Every client query containing a valid bearer token will be handled with the privileges of a Hasura role 'user'. Hasura natively supports the creation of additional roles via the Web UI, but code changes would be required in the authentication server's `handleResponse` function (Figure 34) and database schema to make any use of it. Implementing this functionality could be a simple task yet allow much greater control of client access to resources, expanding the potential audience of the API while also supporting the principle of least privilege.

```
63 exports.getWebhook = async (req, res, next) => {
64   passport.authenticate('bearer', (err, user, info) => {
65     if (err) { return handleResponse(res, 401, {'error': err}); }
66     if (user) {
67       handleResponse(res, 200, {
68         'X-Hasura-Role': 'user',
69         'X-Hasura-User-Id': `${user.id}`
70       });
71     } else {
72       handleResponse(res, 200, {'X-Hasura-Role': 'anonymous'});
73     }
74   })(req, res, next);
75 }
```

Figure 34. Node authentication server's response handler function

Authentication security

A bearer token is nothing more than a string used to verify access, and it's usually passed in the authorization header of every request. This approach has a potential security risk in case the token gets stolen. Switching to JWTs would allow encoding and verifying the tokens via signing, with a set expiry date for each token. Clients could then reach out to the `/login` route to fetch a new token with their user credentials.

Secure communication

HTTP was used for all communication over the web during the project, meaning that all data was transferred in plain text. This means that a potential attacker who has the possibility to intercept the requests, will gain privileged access by being able to read the authentication tokens, for example. Implementing communication over HTTPS requires certificates, which for the employer will be an easy task as trusted certificates already exist and may be utilized to implement this highly secure, encrypted version of HTTP communication.

Logging improvements

As discussed in chapter 8.3.6, Hasura's logging functionality is quite comprehensive by default and can be further adjusted by modifying startup parameters of the Hasura container. Logging could be further improved by hooking the engine container to an external solutions such as Logstash & Elasticsearch, which provide a user friendly, central data storage for logging data with powerful search functionality and analytics tools.

More query definitions

Additional GraphQL queries could be designed and published as HTTP endpoints, making it much simpler for clients to perform frequently repeated complex operations with very simple requests. As shown during implementation, Hasura's HTTP endpoints support dynamic variables, which can be utilized to allow one query to serve multiple clients with different needs, without having to interact with the GraphQL query language at all.

References

A Comprehensive Guide to GraphQL with Hasura. N.d. Informational page on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/graphql/>

Analyzing Query Plans. N.d. Article on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/learn/graphql/hasura-advanced/performance/3-analyze-query-plans/>

Au-Yeung, J. & Donovan, R. 2020. Best practices for REST API design. Article on stackoverflow.blog website. Accessed on 25 September 2021. Retrieved from <https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/>

Authentication & Authorization. N.d. Official documentation on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/docs/latest/graphql/core/auth/index.html>

Brito, G. & Valente, M. 2020. REST vs GraphQL: A Controlled Experiment. Accessed on 4 November 2021. Retrieved from https://www.researchgate.net/profile/Gleison-Brito/publication/339413273_REST_vs_GraphQL_A_Controlled_Experiment/links/5e5001a992851c7f7f4c9bcb/REST-vs-GraphQL-A-Controlled-Experiment.pdf

Brooks, G. 2013. Benefits of APIs. Article on digital.gov website. Accessed on 27 September 2021. Retrieved from <https://digital.gov/2013/03/12/benefits-of-apis/>

Bryman, A. 2012. Social Research Methods. Accessed on 24 October 2021. Retrieved from https://www.academia.edu/38228560/Alan_Bryman_Social_Research_Methods_4th_Edition_Oxford_University_Press_2012_pdf. Oxford University Press

Bush, T. 2020. GraphQL vs REST: The Consumer's Preference. Article on nordicapis.com website. Accessed on 1 November 2021. Retrieved from <https://nordicapis.com/graphql-vs-rest-the-consumers-preference/>

Byron, L. 2015. GraphQL: A data query language. Article on the official engineering.fb.com website, owned by Facebook. Accessed on 17 September 2021. Retrieved from <https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/>

Converis. N.d. Converis. Sales page on the official clarivate.com website. Accessed on 4 October 2021. Retrieved from <https://clarivate.com/webofsciencegroup/solutions/converis/>

CSC. N.d. VIRTAs Publication Information Service. Informational page on official CSC.fi website. Accessed on 24 October 2021. Retrieved from <https://www.csc.fi/en/-/virta-publication-information-service>

Devarkonda S. 2018. Tweaking GraphQL performance using Postgres' Explain command. Article on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/blog/tweaking-graphql-performance-using-postgres-explain-command-6a3d84fd9c9a/>

Doerrfeld, B. 2017. Continuous Versioning Strategy for Internal APIs. Article on nordicapis.com website. Accessed on 1 November 2021. Retrieved from <https://nordicapis.com/continuous-versioning-strategy-for-internal-apis/>

EuroCRIS. 2014. CERIF in Brief. Archived page from the eurocris.org website. Accessed on 4 October 2021. Retrieved from https://www.eurocris.org/eurocris_archive/cerifsupport.org/cerif-in-brief/index.html

European Commission. 1991. Commission Recommendation concerning the harmonisation within the Community of research and technological development databases. A recommendation to member states of the EU. Accessed on 4 October 2021. Retrieved from <https://wayback.archive-it.org/12090/20161116062703/http://cordis.europa.eu/pub/cerif/docs/cerif1991.htm>

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine. Accessed on 9 November 2021. Retrieved from <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

Frequently Asked Questions (FAQ). N.d. Answers to questions on the official GraphQL website. Accessed on 17 September 2021. Retrieved from <https://graphql.org/faq/>

Get started with JSON Web Tokens. N.d. Article on Auth0 website. Accessed on 4 November 2021. Retrieved from <https://auth0.com/learn/json-web-tokens/>

Gilling, D. 2021. REST vs GraphQL APIs, the Good, the Bad, the Ugly. Article on moesif.com website. Accessed on 7 November 2021. Retrieved from <https://www.moesif.com/blog/technical/graphql/REST-vs-GraphQL-APIs-the-good-the-bad-the-ugly/>

GraphQL schema basics. N.d. Article on Apollo Docs website. Accessed on 11 November 2021. Retrieved from <https://www.apollographql.com/docs/apollo-server/schema/schema/>

Gupta, L. 2021. Statelessness in REST APIs. Article on restfulapi.net website. Accessed on 11 November 2021. Retrieved from <https://restfulapi.net/statelessness/>

Hagen, N. 2018. 3 Methods to resolve GraphQL endpoints. Article on contentful.com website. Accessed on 11 November 2021. Retrieved from <https://www.contentful.com/blog/2018/09/25/3-methods-resolve-graphql-endpoints/>

Harguindeguy, B. 2021. Everything You Need to Know about API Security in 2021. Article on pingidentity.com website. Accessed on 16 November 2021. Retrieved from <https://www.pingidentity.com/en/company/blog/posts/2020/everything-need-know-api-security-2020.html>

Hasura Products: Cloud, Open Source & Enterprise (On-Prem). N.d. A sales page on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/products/>

How Hasura GraphQL engine works. N.d. Article on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/docs/latest/graphql/core/how-it-works/index.html>

Instant GraphQL and REST for Databases. N.d. Informational page on hasura.io website. Accessed on 18 November 2021. Retrieved from <https://hasura.io/graphql/database/>

Jacobs, I. 2004. URIs, Addressability, and the use of HTTP GET and POST. W3C Technical Architecture Group 21 March 2004. Accessed on 17 November 2021. Retrieved from <https://www.w3.org/2001/tag/doc/whenToUseGet.html>

Jain, A. 2019. REST — Stop calling your HTTP APIs as RESTful APIs (Part 1). Article on medium.com website. Accessed on 11 November 2021. Retrieved from <https://medium.com/the-sixt-india-blog/rest-stop-calling-your-http-apis-as-restful-apis-e8336e3e799b>

Kananen, J. 2015. Opinnäytetyön kirjoittajan opas : Näin kirjoitan opinnäytetyön tai pro gradun alusta loppuun. Jyväskylä: Jyväskylän ammattikorkeakoulu. Accessed on 2 October 2021

Lukka, K. 2003. The Constructive Research Approach. Accessed on 1 December 2021. Retrieved from https://www.researchgate.net/publication/247817908_The_Constructive_Research_Approach, Publications of the Turku School of Economics and Business Administration

Madden, N. 2020. API Security in Action. Manning. Accessed on 20 September 2021. Retrieved from <https://janet.finna.fi/>, Skillsoft Books ITPro.

Momjian, B. 2001. PostgreSQL. Addison-Wesley. Accessed on 31 October 2021. Retrieved from http://www.foo.be/docs-free/aw_pgsql_book.pdf

Nikkanen, J. & Schirrwagen, J. 2019. Providing Finnish national Publication Data to OpenAIRE – Case VIRT.A. OpenAIRE blog. Accessed on 24 October 2021. Retrieved from <https://www.openaire.eu/virta-finnish-national-publication-data-to-openaire>

Oggier, C. 2020. How fast GraphQL is compared to REST APIs. Bachelor's Thesis from Haaga-Helia University of Applied Sciences. Accessed on 4 November 2021. Retrieved from https://www.the-seus.fi/bitstream/handle/10024/340318/Thesis_Camille_Oggier.pdf

OpenAIRE. 2017. CRIS information elements relevant for OpenAIRE. Official guidelines provided by OpenAIRE. Accessed on 14 October 2021. Retrieved from https://openaire-guidelines-for-cris-managers.readthedocs.io/en/latest/cris_elements_openaire.html

OpenAIRE. N.d. What is OpenAIRE and what are OpenAIRE services?. Answers to questions on the official OpenAIRE website. Accessed on 14 October 2021. Retrieved from <https://www.openaire.eu/faqs>

Open Science EU. 2020. OpenAIRE (Open Access Infrastructure for Research in Europe). Article on openscience.eu website. Accessed on 14 October 2021. Retrieved from <https://openscience.eu/openaire/>

Pearlman, S. 2016. What are APIs and how do APIs work? Article on mulesoft.com website. Accessed on 25 September 2021. Retrieved from <https://blogs.mulesoft.com/learn-apis/api-led-connectivity/what-are-apis-how-do-apis-work/>

Pratt, M. 2021. Difference Between REST and HTTP. Article on baeldung.com website. Accessed on 11 November 2021. Retrieved from <https://www.baeldung.com/cs/rest-vs-http>

Reiser, A. 2018. Why HATEOAS is useless and what that means for REST. An article on medium.com website. Accessed on 11 November 2021. Retrieved from <https://medium.com/@andreasreiser94/why-hateoas-is-useless-and-what-that-means-for-rest-a65194471bc8>

Requests and Responses. N.d. Article on dgraph website. Accessed on 17 November 2021. Retrieved from <https://dgraph.io/docs/graphql/api/requests/>

Schultz, M. 2018. Unifying your Data with GraphQL. Article on envylabs.com website. Accessed on 16 November 2021. Retrieved from <https://blog.envylabs.com/unifying-your-data-with-graphql-6a039d056fc4>

Sikandar, A. N.d. GraphQL vs. REST: What you didn't know. Article on mobilelive.ca website. Accessed on 4 November 2021. Retrieved from <https://www.mobilelive.ca/blog/graphql-vs-rest-what-you-didnt-know/>

Simha D. 2019. Authentication and Authorization in GraphQL (and how GraphQL-Modules can help). Article on the-guild.dev website. Accessed on 1 November 2021. Retrieved from <https://the-guild.dev/blog/graphql-modules-auth>

Stemmler, K. 2020. 4 Simple Ways to Call a GraphQL API. Article on apollographql.com website. Accessed on 17 November 2021. Retrieved from <https://www.apollographql.com/blog/graphql/examples/4-simple-ways-to-call-a-graphql-api/>

Stuart M. 2018. GraphQL Resolvers: Best Practices. Article on medium.com website. Accessed on 15 November 2021. Retrieved from <https://medium.com/paypal-tech/graphql-resolvers-best-practices-cd36fdbcef55>

Thelin, R. 2021. What are REST APIs? HTTP API vs REST API. Article on educative.io website. Accessed on 22 September 2021. Retrieved from <https://www.educative.io/blog/what-are-rest-apis>

Walkowski, D. 2020. Securing APIs: 10 Ways to Keep Your Data and Infrastructure Safe. Article on f5.com website. Accessed on 1 November 2021. Retrieved from <https://www.f5.com/labs/articles/education/securing-apis--10-best-practices-for-keeping-your-data-and-infra>

Webb, J. 2015. Consumers Demand More Visibility Into The Supply Chain. Article on Forbes.com website. Accessed on 1 November 2021. Retrieved from <https://www.forbes.com/sites/jwebb/2015/10/26/consumers-demand-more-visibility-into-the-supply-chain/?sh=e27acd95b703>

Wieruch, R. 2019. The Road to GraphQL. Accessed on 17 September 2021. Retrieved from <http://leanpub.com/the-road-to-graphql>

Appendices

Appendix 1. The final GetPublication query

GraphiQL

 Prettyfy History Copy Explorer Code Exporter Voyager REST Derive action Analyze

```

1 * query GetPublication($id: BigInt!, $fields: [String!]!){
2   dataentity(
3     where: { typename: { _eq: "Publication" }, _and: { id: { _eq: $id } } }
4   ) {
5     string_attributes (
6       where: {
7         attributedefinition: {
8           name: { _in: $fields }
9         }
10      }
11    ){
12      stringValue
13      attributedefinition {
14        name
15      }
16    }
17    number_attributes (
18      where: {
19        attributedefinition: {
20          name: { _in: $fields }
21        }
22      }
23    ){
24      numberValue
25      attributedefinition {
26        name
27      }
28    }
29  }
30 }

```

Query definition

Variables passed to test the query

```

1 {
2   "id": 7906687,
3   "fields": ["publYear", "cfitle", "fissue", "referencePlain"]
4 }

```

```

* {
*   "data": {
*     "dataentity": [
*       {
*         "string_attributes": [
*           {
*             "stringValue": "Testiartikkeli",
*             "attributedefinition": {
*               "name": "cfitle"
*             }
*           },
*           {
*             "stringValue": ""
*             "attributedefinition": {
*               "name": "fissue"
*             }
*           },
*           {
*             "stringValue": "Oysti, S. (2021). Testiartikkeli. In S. Oysti (Ed.), Testiartikkeli
https://doi.org/10.1177/0002716208331379",
*             "attributedefinition": {
*               "name": "referencePlain"
*             }
*           }
*         ],
*         "number_attributes": [
*           {
*             "numberValue": "2021.000000000",
*             "attributedefinition": {
*               "name": "publYear"
*             }
*           }
*         ]
*       }
*     ]
*   }
* }

```

API's full response to the test query