

Zhanna Kreteva

USING MODERN GRAPHQL SOLUTIONS IN THE CONTEXT OF INTERNET OF THINGS

USING MODERN GRAPHQL SOLUTIONS IN THE CONTEXT OF INTERNET OF THINGS

Zhanna Kreteva
Bachelor's Thesis
Autumn 2020
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Bachelor's Degree in Information technology

Author(s): Zhanna Kresteva

Title of Bachelor's thesis: Using Modern GraphQL Solutions in Context of Internet of Things

Supervisor(s): Kari Laitinen

Term and year of completion: Autumn 2020

Number of pages: 53

The Internet of Things industry has in many ways become vital to access a new step in modern technological progress. With an estimation of more than 20 billion interconnected devices in the world, there are endless reasons varying from personal use to industrial, business and scientific innovations for careful optimization and management of those devices. Haltian, a modern Internet of Things startup, offers full-stack solutions for the clients: from creating hardware (sensors, devices), implementing databases and application programming interfaces, to creating desktop, mobile and web applications.

Haltian Oy is using representational state transfer application programming interfaces and has been facing architectural and performance issues due to the increasing size of the said interfaces. To solve the issues, it was decided to test a different approach that is preferred to RESTful architecture by many software companies. The approach is to utilize GraphQL, a modern query language, to rebuild one of the existing APIs that is implemented and deployed using AWS Serverless and AWS Relational PostgreSQL database.

New experimental GraphQL API was built in compliance with the existing technologies used by the company: an Apollo server is deployed to AWS Lambda, and the endpoint is exposed via API Gateway. The testing of the functionality has proven a success of the solution in agile and customizable data querying, reduction of the number of endpoints by 95.5% and has been reviewed enthusiastically by the developers at the client company.

The decision was made to continue working on the implementation of the API further, by optimizing the relational database schemas to perform in hand with the new GraphQL solution.

Keywords:

Internet of Things, Amazon Web Services, GraphQL, API, PostgreSQL

CONTENTS

| | |
|--|----|
| TERMS AND ABBREVIATIONS | 5 |
| 1 INTRODUCTION | 6 |
| 2 OVERVIEW OF CURRENT TECHNOLOGIES USED BY HALTIAN | 8 |
| 2.1 Serverless framework | 10 |
| 2.1.1 API Gateway | 11 |
| 2.1.2 AWS Lambda | 12 |
| 2.1.3 Amazon relational database | 13 |
| 2.2 Disadvantages of the current technologies | 14 |
| 3 GRAPHQL | 15 |
| 3.1 What is GraphQL? | 15 |
| 3.2 Schema | 17 |
| 3.3 Types and resolvers | 18 |
| 3.4 Queries and Mutations | 20 |
| 3.5 Comparing GraphQL API to REST API | 23 |
| 4 IMPLEMENTATION (ACTUAL HALTIAN API OPTIMIZATION) | 26 |
| 4.1 Structure of the current API | 26 |
| 4.2 Architecture of a new GraphQL solution | 27 |
| 4.3 Developing the solution | 28 |
| 4.3.1 Creating the Apollo server | 28 |
| 4.3.2 Defining the schema | 29 |
| 4.3.3 Adding data sources | 32 |
| 4.3.4 Adding resolvers | 33 |
| 4.4 Deployment | 34 |
| 4.5 Testing | 37 |
| 4.5.1 Get all devices | 38 |
| 4.5.2 Get device and its messages by device identifier | 39 |
| 4.5.3 Get all messages | 41 |
| 4.5.4 Get messages by message identifier | 42 |
| 5 EVALUATION | 44 |
| 6 CONCLUSION | 45 |
| REFERENCES | 46 |

TERMS AND ABBREVIATIONS

| | |
|------|---------------------------------|
| API | Application Program Interface |
| REST | Representational state transfer |
| EC2 | Elastic Compute Cloud |
| IoT | Internet of things |
| AWS | Amazon Web Services |
| HTTP | Hypertext Transfer Protocol |
| SDL | Schema Definition Language |
| BLE | Bluetooth Low Energy |

1 INTRODUCTION

This thesis was written at the supervision of Haltian Oy, an international internet of things company, that specializes in innovative electronic product development and internet of things (IoT) solutions created with the company's own Thingsee IoT solution platform. The platform includes hardware (Thing see sensors and gateways), software (Thingsee Solution Suite), solutions based on Amazon Web Services and other cloud tools that allow to create scalable and efficient solutions for various enterprises.

As Internet of Things refers to the physical devices that are able to communicate data with the network ^[1], it is important to ensure the most sufficient information transfer. Just like in any other dynamic software application, data communication in IoT web applications plays a crucial role, due to both the amount of data as well as the number of connected devices (or “things”) that generate it. As the data is requested, modified, added – application data flow emerges between the layers of the application. Depending on the purpose of the application it may have different architectural layers, that are grouped into software design patterns. Elements in a software design patterns are logical representatives of how data is transferred to and from the user.^[2] For example, MVC pattern in a web-based application usually presumes to include Model as a first layer of interaction with a database ^[3], an Application Programming Interface or an API as the Controller layer and user interface as the View.

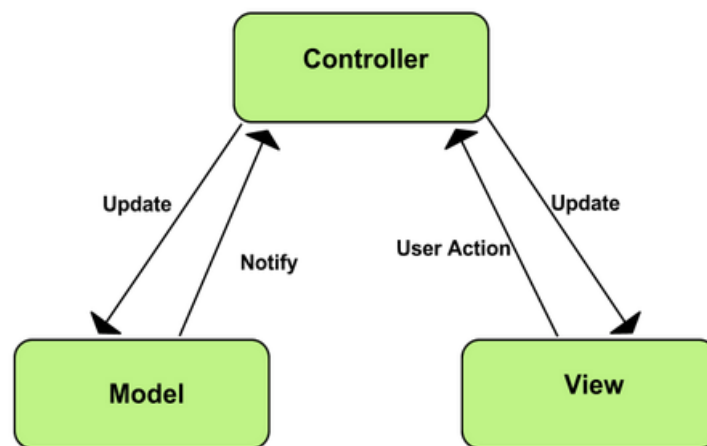


FIGURE 1. Data flow between components of MVC.^[4]

In that scenario, model layer interacts with a database while retrieving or modifying the information using queries like select, insert, update, and delete [5]. The view layer would take an input from a user, in a form of a request, command or data. The controller layer communicates this user input to the model, and transfers needed output back to the view. Improving the data flow between these layers in web applications often means considering new approaches to deliver both the model, controller and view layer. That can mean looking at how new technologies perform in comparison to older tools, as well as taking actual software development experience into account.

According to Samuel Buna [6], relational databases already “deliver on reliability, consistency and integrity for the task of storing data”. It is the current solutions for communication data between the layers of the application that have many problems. Representational state transfer or REST APIs and ad hoc HTTP endpoints, which are a status quo in many web applications, are often considered not optimal enough due to a number of weaknesses including “multiple roundtrips” and over-fetching [7]. First weakness refers to the need to make multiple “roundtrips” between server and client while retrieving complicated object graphs to render single views. Second weakness, or over-fetching, occurs when more data is fetched than it is needed.

Haltian Oy is using representational state transfer application programming interfaces and has been facing architectural and performance issues due the increasing size of the said interfaces. To solve the issues, it was decided to test a different approach that is preferred to RESTful architecture by many software companies, including Facebook, Instagram and Twitter [8]. The approach is to utilize GraphQL, a modern query language, to rebuild one of the existing APIs that is implemented and deployed using AWS Serverless and AWS Relational PostgreSQL database.

Conducting a thorough comparative study and integrating GraphQL solutions into an internet of things company's set of provided services is the primary aim of this research. In the process of developing of the new API, the efficiency and quality of GraphQL deliverables will be assessed and contrasted to the existing tools. The supporting hypothesis of the research expects performance and architectural optimization of the API through avoiding the pitfalls of REST APIs such as over-fetching and escalation of the number of the ad hoc endpoints.

2 OVERVIEW OF CURRENT TECHNOLOGIES USED BY HALTIAN

Haltian, a modern Internet of Things company, offers full-stack solutions for the clients: from creating hardware (sensors, devices), implementing databases and application programming interfaces, to creating desktop, mobile and web applications ^[9]. Haltian provides world class IoT hardware and cloud solutions under the name “Thingsee”, that includes Thingsee IoT sensor and gateway devices, Thingsee Solutions Suite of IoT management applications, and cloud solutions, such as Thingsee Operations Cloud ^[10]. The core of the Thingsee platform is the various wireless Bluetooth Low Energy (BLE) or mesh network sensors ^[11] that are made to collect all the required data for various IoT use cases: temperature, humidity, air quality, barometric pressure, occupancy, distance and more. Depending on the use case, the sensor may contain internal analytics of the data or the ability to store data for longer measurement periods. As a default, Thingsee sensors use Wirepas Mesh - a wireless connectivity technology for massive IoT ^[12]. Wirepas Mesh is considered a feasible solution for cases when power consumption is critical, and real-time data transfer or high data bandwidth are not required. BLE sensors can be used for cases where real-time communication or bandwidth requirements are higher than it is feasible with Wirepas Mesh ^[11].

Following this, IoT Gateway is a hardware device that takes on a key role of enabling device-to-device communication (between the Thingsee devices) or device-to-cloud communication (Thingsee sensor devices to Thingsee Operations Cloud) ^[13], ^[14]. In the Thingsee platform, sensors are connected to a Thingsee Gateway device. Thingsee Gateway verifies working connectivity between the sensors and the cloud and allows to manage and monitor the local sensor network. In addition, gateway operates maintenance tasks such as firmware updates, sensor and network diagnostics, and recovery operations ^[14].

Thingsee Operations Cloud is the underlying infrastructure needed for processing and storing IoT data received from the gateway. Thingsee IoT Platform is built upon Amazon Web Services. All operations are implemented using Serverless framework to achieve scalability and cost-effectiveness regardless of the number of sensors. Thingsee Operations installations (“Thingsee Operations Profiles”) are connected to the global database to ensure that all manufactured Thingsee devices, certificates, and additional inventory data are transferred to the specific client

profile as intended. Thingsee Operations Edge, on the contrary, is the on-premises version of the API that carries through a subset of the features [15].

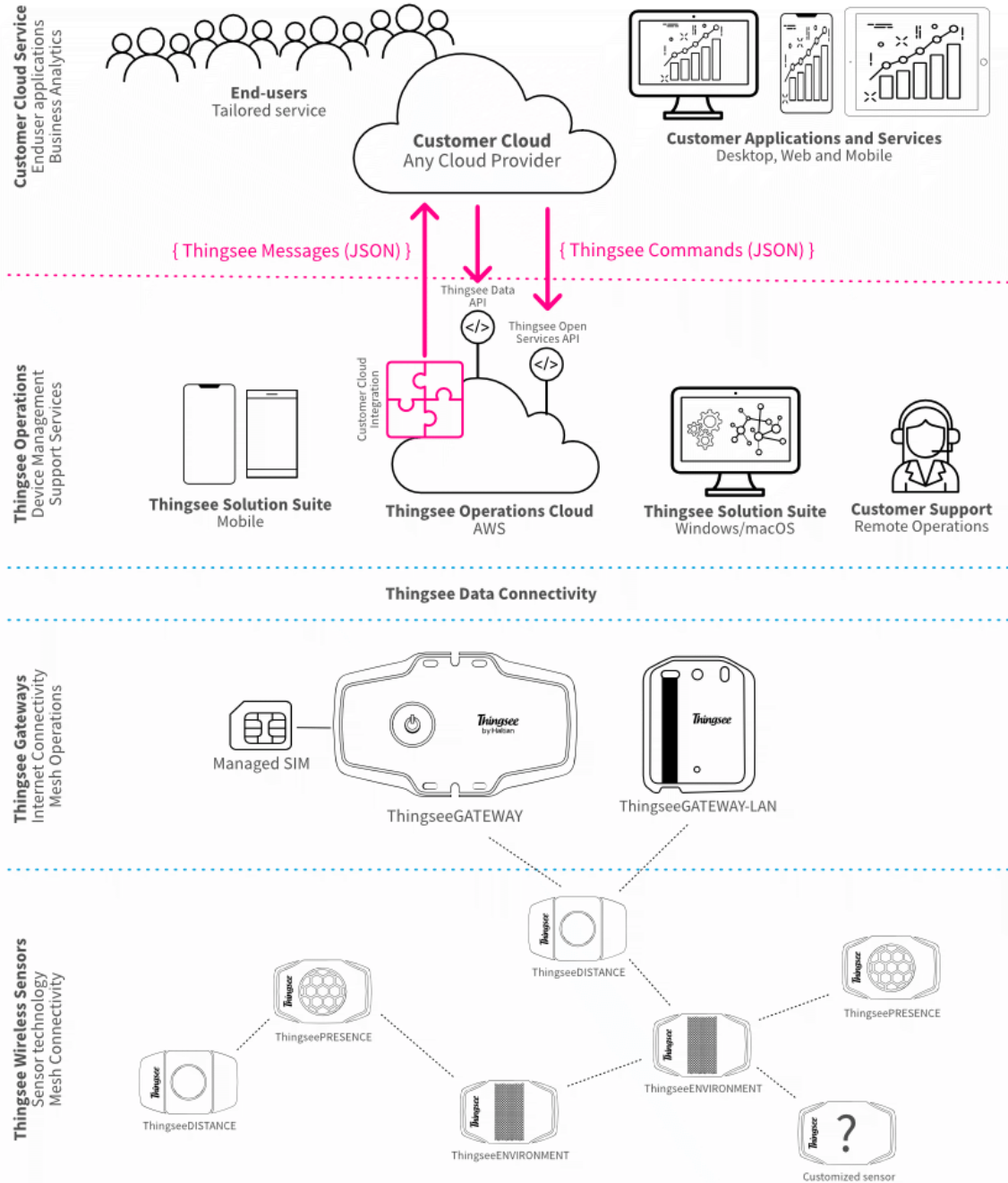


FIGURE 2. Thingsee IoT as a Service [16].

Data communication from the Thingsee sensors the Customer cloud through the Gateway and Operations Cloud is illustrated in Figure 2. As evidenced on the diagram, one of the tools for data transfer between the Thingsee Operations cloud to the customer platform is apprehended by the

Thingsee Open Service API. The purpose of the Open API is to provide company customers with the tools to control sensor data or to create individualized implementation for data management in the context of Internet of Things. This API is a public version of the Haltian's Service API which is dedicated exclusively for internal use. Thingsee Open Service API is included in both Thingsee Operations Cloud and Thingsee Operations Edge. Use cases for the Open API include creating a device management dashboard, a field service application or can be utilized to connect Thingsee device inventory data to customer's asset management tools ^[17].

To achieve scalability and effective billing system, as well as minimize server maintenance – Thingsee Open API is built upon Serverless framework, which is an innovative solution for many use-cases, including scalable APIs ^[18]. A common structure of a Serverless stack includes an HTTP gateway service, a computing service and a database service ^[19]. In the context of Thingsee Open API – AWS API Gateway acts as a gateway service, AWS Lambda as a computing service and AWS Relational Database as a database service. Experimental part of this research is composed of integrating GraphQL technologies into Haltian's Thingsee Open Service API to replace current technologies. Next subsections will characterize present-day services in detail as well as investigate drawbacks that Haltian has encountered with these technologies.

2.1 Serverless framework

Serverless computing is a new approach in web-development that allows developers to shift focus from infrastructure management (scaling and maintenance) to originating functionality that brings substantial value ^[20]. Result of this approach is a possibility to engage in the innovative development process that is more agile in response to changes. Serverless framework offers services like AWS Lambda that are natively integrated into AWS, and follow four key properties: auto-scaling, zero administration, increased velocity and pay-per-use billing model ^[21]. As depicted in [22]:

- Auto-scaling refers to automatic handling of scaling challenges depending on fluctuations of the traffic.
- Zero administration property allows to minimize maintenance time on things as EC2 instances, fleets and even operating systems.
- Increased velocity of the development allows for faster and more effective results and testing.

- Pay-per-use billing model ensures that all the function-as-a-service compute and managed services are charged based on usage rather than pre-provisioned capacity. This results with up to 90% reduction of the expenses compared to cloud Virtual Machine.

As Thingsee Open API relies on a Serverless stack structure, it is important to break down three layers of the services that were utilized in development of this RESTful API.

2.1.1 API Gateway

Thingsee Open Service API utilizes API gateway as a gateway service layer of the Serverless stack. It is adhered to a REST architectural constraint. Amazon API Gateway is one of the AWS services that allows to create, publish, monitor, secure and maintain web-based APIs (application programming interfaces) regardless of the scale of the application [23]. Cloud applications can access data, business logic, and functionality from the backend services through the API. API Gateway use-cases described in AWS Documentation include: creation of RESTful APIs and WebSocket APIs with real-time two-way communication, support of containerized and serverless workloads, and web applications [23].

In Thingsee Open API - API Gateway handles all the tasks involved in accepting and processing around thousands of concurrent API calls. Other features that are useful for Thingsee Platform include traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management [24]. API Gateway has a specific fee policy, where the total sum of payment is calculated according to the number of the received API calls and the amount of transferred data.

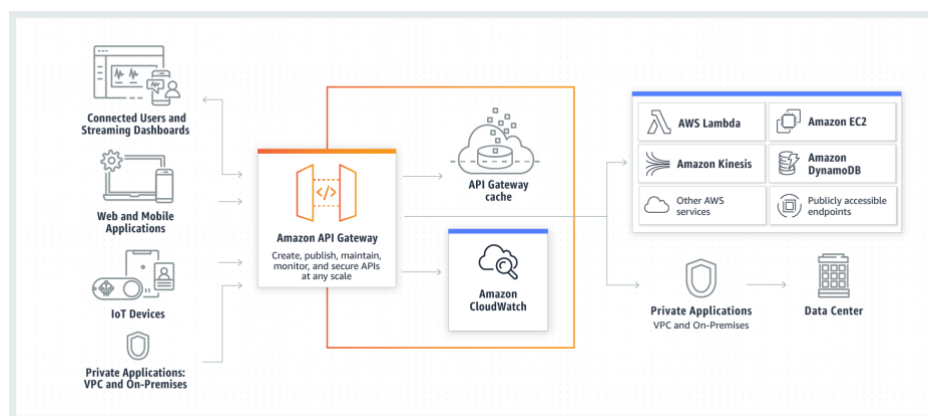


FIGURE 3. API Gateway architecture [25].

This diagram above illustrates how the data from the Thingsee sensor and gateway devices gets transferred to the Amazon API Gateway. Thingsee Open APIs that is built in Amazon API Gateway provide Haitian developers with a consistent developer experience for further AWS Serverless application development. Amazon API Gateway routes different parts of the API to different AWS Lambda functions. Data from the API Gateway is furthermore analysed by careful examination of logs at Amazon CloudWatch and is managed by upper layers of Thingsee Operations Cloud, including data routing, fetching with AWS Lambda cloud functions and extensions through the Serverless framework and data storage with Amazon Relational Database service.

2.1.2 AWS Lambda

AWS Lambda, which is a serverless computing service provided by Amazon Web Services (AWS) is used as a computing service [26]. Haitian developers create functions written in one of the supported languages and runtimes depending on the client, and upload them to AWS Lambda, which executes those functions. As characterized in Serverless Documentation [26], when a function is created, Lambda packages it into a container, which is assigned necessary RAM and CPU capacity before the execution starts. The execution is event-triggered, which in the context of Haitian usually means: an AWS API Gateway HTTP endpoint request, a MQTT broker event or AWS SNS topic message. After the function is finished running, the RAM allocated at the beginning is multiplied by the amount of time the function spent running. The charges are then calculated based on the memory allocated for the function and the amount of run time it took to finish execution.

Figure 3 illustrates the efficiency of AWS Lambda computing service from the development perspective. In Thingsee Open API one execution of a Lambda function serves a single HTTP request, with the different parts of the API routed to different Lambda function via Amazon API Gateway. AWS Lambda is auto scalable, depending on the demand different parts of Open API are scaled independently according to the ongoing usage proportions allowing for a flexible setup.

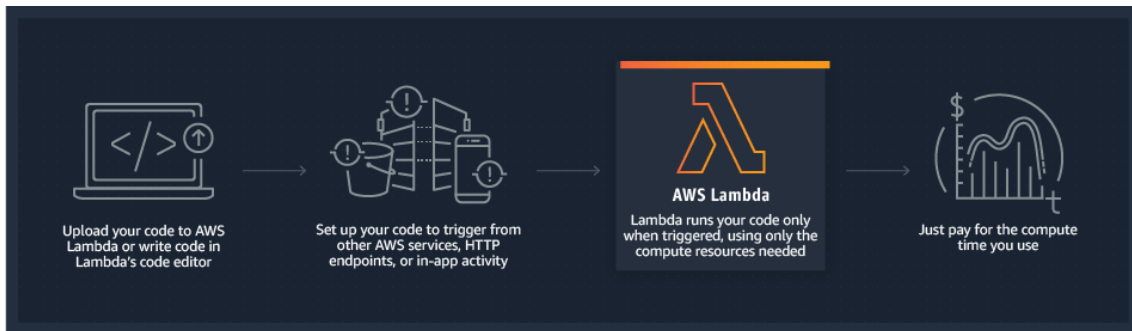


FIGURE 3. AWS Lambda. How it works [27].

2.1.3 Amazon relational database

Amazon Relational Database Service or Amazon RDS serves as a database service in Thingsee Open API. The service provides flexible capacity while managing database setup, backups, software patching, automatic failure detection, recovery [27]. DB instance is the building block of Amazon RDS that runs a database engine. Thingsee Open API utilizes PostgreSQL as the database engine. PostgreSQL is an open-source relational database system, that supports both SQL (relational) and JSON (non-relational) querying [28]. Amazon RDS is used to set up, operate, and scale PostgreSQL deployments in the Thingsee Operations Cloud.

Deployed AWS Lambdas that are routed to the certain HTTP endpoint of the Thingsee Open API access the database instance through the 'db-connector' module with the credentials defined in the template as global variables. Depending on the HTTP request (GET, POST, PUT, DELETE) send by the Lambda, data in the database can be fetched, mutated or destroyed. The request is performed by an SQL query, that is unique for each request.

```
const dbConnector = require('db-connector');

let client;
const clientOpts = {
  user: process.env.RDB_USER,
  password: process.env.RDB_PASSWORD,
  database: process.env.RDB_DB,
  host: process.env.RDB_HOST,
  port: process.env.RDB_PORT,
};
```

FIGURE 4. "dbConnector" module in Thingsee Open API Lambda function.

2.2 Disadvantages of the current technologies

When assessing the disadvantages of the technologies utilized by the Thingsee Open API it is important to account for the specific circumstances that may bring the weaknesses of the said technologies forward. Some known flaws in the technologies might never become a measurable drawback in one context, but in the context of Hiltian and IoT certain technological defects put cost, maintenance or the development process at a disadvantage. One of the circumstances is the scale of data communication, both from the devices to the cloud, and querying to the API. With some customers running less than 10 sensors in their stack to thousands of devices in other customer's use case, it became more difficult to maintain backward compatibility of the Thingsee Open API. A new version of the API would have minimal changes of the current state API calls in order to maintain those for the customers. However, the new version of the Thingsee Open API has become a necessary step, for the reasons elucidated further in this section.

According to current Hiltian cloud developers, with the increasing amount of data, many API calls have become more complex, requesting more relational data in a query due to under-fetching issue that comes with RESTful architecture. That makes the corresponding SQL query more extensive, taking longer period of time to finish AWS Lambda execution, making it more memory consuming and therefore less cost-efficient. A development requirement in the Thingsee platform is setting Lambda timeout to eight seconds, as longer time results in a higher cost without adding much value for the optimization and debugging. A Lambda that took more than that set time to execute exits with the "Timeout error", without a response to the complex query. Another performance issue – over fetching, occurs when simple queries get a fixed response, which contains more data than requested.

Lastly, the API has become more complex with the increasing number of operations and endpoints. Documentation of the API is difficult to manage, due to both the amount of the endpoints and not-optimized API versioning. Some customers have an older version of the API, and some the newest. The goal of the experimental research would be minimizing the number of endpoints, eliminating over and under fetching issues as well as optimizing API versioning. In order to address these challenges at once, it was decided that a modern GraphQL language will be utilized to build a new Thingsee Open API.

3 GRAPHQL

3.1 What is GraphQL?

Technopedia^[30] defines a query language as any computer programming language that requests and retrieves data from a database and information systems via sending user entered structured command-based queries. One of such query languages, GraphQL, was released by Facebook in 2016 to address several problems faced by developers when using standard architectural styles such as Representational State Transfer^[31]. This chapter will cover definition, specifications and use cases of GraphQL in modern development of web-based APIs as well as a compare it with RESTful APIs.


First and foremost, GraphQL is both a query language for web APIs and a corresponding runtime for executing those queries^[32]. As Samer Buna explains in his book “Learning GraphQL and Relay”, it is important to understand both parts of the above definition. If a software client application and backend data service both “speak” GraphQL, data requirements can be communicated “declaratively”. Secondly, it can be used as an execution layer, or a GraphQL speaking “translator”, that understands and responds to requests made with GraphQL language. In the context of web development, programmers can use this novel query language for writing client requests, runtimes for server applications, and, as will be attempted in this research: for creation of web-based APIs. Common tasks of an Application Programming Interface or an API include:

- Acting as a controller between protected services and software clients
- Processing raw data into client specified structure
- Constructing SQL joint statements to respond efficiently to a client request
- Returning data in specific formats such as JSON or XML^[33].

Before implementing the functionality described above in a GraphQL API, it is important to understand nuances and standards of GraphQL development. Design of a GraphQL API starts with a schema, that includes all data types, relations, and possible operations on that data^[31]. Schema is written with Schema Definition Language (SDL) and serves as a contract between the client and the server to define how a client can access the data. Strong typing system ensures

that the query is both syntactically correct and valid within the GraphQL type system before execution, and it provides insightful error handling with the resolvers [34]. As promised by the creators at GraphQL.org [32]: “Ask for what you need, get exactly that”, meaning that a server will respond with the exact data that was requested in a client query. Query often mirrors the shape of its JSON response, as they “speak” the same format. As seen in Picture 2, a GraphQL query has the same structure as a response, just the values are missing. In the small books API that is used for this example, a book object has an id, a name and an author. A query below asks only for the names of all books, and response return exactly that data without the extra fields.

These features of GraphQL not only improve development process and help to solve the over-fetching and under-fetching issue, but also allow to detect deviations and errors by pointing into the exact query fields that are not resolving correctly. Measuring performance of a particular resolver function can also provide crucial insights into the issues of the system [34].



```
1 {
2   books{
3     name
4   }
5 }
```

```
1 {
2   "data": {
3     "books": [
4       {
5         "name": "Harry Potter and the Chamber of Secrets"
6       },
7       {
8         "name": "Harry Potter and the Prisoner of Azkaban"
9       },
10      {
11        "name": "Harry Potter and the Goblet of Fire"
12      },
13      {
14        "name": "The Fellowship of the Ring"
15      },
16      {
17        "name": "The Two Towers"
18      },
19      {
20        "name": "The Return of the King"
21      },
22      {
23        "name": "The Way of Shadows"
24      },
25      {
26        "name": "Beyond the Shadows"
27      }
28    ]
29  }
30 }
```

FIGURE 5. GraphQL query and response

While fetching the data to satisfy a client request with a RESTful API it is typical to access multiple endpoints. In the case of GraphQL only one endpoint is exposed, where all the requests can be responded to, which has both advantages and disadvantages. Before continuing with the comparison of REST and GraphQL APIs, it is important to cover technical specifications and thorough definitions of GraphQL API's core components: what is schema and how to define it,

typing system and resolver functions, queries and mutations as well as additional tools that would allow to implement GraphQL in the context of cloud development.

3.2 Schema

A GraphQL implementation is defined by a multi-graph schema ^[35], which is written using the GraphQL Schema Definition Language or SDL. SDL is an official part of the GraphQL specification, and is considered to be the most comprehensive way to define a schema. The syntax of a schema is human readable and resembles JavaScript, however it is not linked to any programming language, or in other words is “language-agnostic” ^[36]. Multi-graph in this context refers to the logical data model of the schema, where each object node has a type and list of fields; each field has a name and also a type ^[31]. Understanding the data model may seem complicated, but it is as simple as describing data in a spoken language. For example: each book will have an id, a title, a description, and the name of the author. This sentence describes a book type that can be seen below:

TABLE 1. GraphQL schema object, query and mutation types.

| Example 1. | Example 2. |
|---|---|
| <pre> type Book { ID: Int Title: String Description: String Author: [Author] } </pre> | <pre> schema { query: Query mutation: Mutation } </pre> |

Types and resolvers will be studied thoroughly in the next section, nevertheless, the Example 1 in the above table shows how simple it is to declare data objects offered by a GraphQL API, and Example 2 shows how the `schema` defines fields query and mutation, where “query” forms the entry (endpoint) for any valid query and “mutation” allows for possible addition, deletion and edit of the data through the API. It is required that a GraphQL schema contains a Query operation type ^[37], as it defines what can be retrieved by a query.

3.3 Types and resolvers

As mentioned previously, GraphQL relies on a strong typing system. This section will cover definitions of the types and resolvers listed in official GraphQL documentation [37]. To comply with the typing rules, it is important to know all the types that can be represented in a GraphQL schema, starting with one of the most fundamental components - the object type. Object type defines an object that can be fetched from an API and its fields, each one also having a definitive type. A field of an Object type can have zero or multiple arguments, that are passed by a specific name, can be required or optional, and can be assigned a default value (if optional) [37].

Scalar types resolve the fields of an Object type that do not have any subfields and are defined as the “leaves of the query”. GraphQL has a set of default scalar types that are listed in Table 2.

TABLE 2. GraphQL scalar types.^[37]

| | |
|--------------------|--|
| Int | a signed 32-bit integer |
| Float | a signed double-precision floating-point value |
| String | a UTF-8-character sequence. |
| Boolean | true or false |
| ID | a unique identifier, serialized in the same way as a String; however, is not intended to be human-readable |
| Custom scalar type | a scalar type, that can be serialized, deserialized, and validated in a custom way |

Enumeration types or *Enums* are scalars, restricted to a particular set of permitted values. Using enumeration types allows to validate that arguments are in the range of those values and to communicate that a field will always be one of a finite set of values. Depending on a programming language of a GraphQL service implementation there are specific ways of utilizing enumeration types. Some implementations can take advantage from programming languages that support *Enums*, in other cases with no *Enum* support such as JavaScript, values could be internally mapped to a set of integers [37].

Object types, scalars, and enums are the only types that can be defined in the process of GraphQL development. However, to affect the validation of the values additional type modifiers can be applied. For example, a type can be marked as Non-Null by adding an exclamation mark after the type name. In that case a server expects to return a Non-Null for a specified field or its argument, otherwise a GraphQL execution error will be triggered. Another kind of type modifier is List, which indicates that a field or an argument of a field will respond with an array of the specified type. This type modifier is denoted by wrapping a type in square brackets. Both of these modifiers can be combined, validating a List of Non-Null types: `[String!]`.

Next is an *Interface*, which is an abstract type that includes a certain set of fields that a type must include to implement the interface. If one creates an interface that includes certain fields and arguments to implement an abstract type, another type that implements the interface must include the exact fields, arguments and return types as defined in the interface. This is similar to how a child class would inherit properties of a parent class that it extends in JavaScript. Union type is a more constricted version of an Interface and has to be used with a conditional fragment, so the query is successful.

In cases such as adding new data through a mutation, Input types are used in GraphQL schema language. Fields of an input are similar to those of an Object type; however, they cannot have arguments. Use of the input types in mutations will be discussed next.

To conclude the definition of types in GraphQL - Query and Mutation types have to be explained. These types are crucial in a schema, as every GraphQL service must include a query type to define the entry point of every GraphQL query. Mutation type would be necessary if the data needs to be added, modified or deleted via the queries. Table 1 on Page 18 shows how these types are attached to a schema.

As for the resolver functions, they are defined as functions by the GraphQL server developer that resolve fields on each type ^[38]. When a field is executed it triggers a resolver function. If a field produces an object value, then the query will contain another selection of fields and the query will continue until scalar values are reached. A resolver function receives four arguments: `obj` (the previous object), `args` (arguments to the field), `context` (contextual information) and `info` (field specific information relevant to the current query).

Resolver can be Asynchronous, when the context argument is used to provide access to a database. Since loading from a database is an asynchronous operation, this requests a Promise. When a resolver returns a list of Promises, it is called a List resolver. Other examples include Trivial resolvers, that are usually omitted due to their simplicity, and Scalar coercion, when a resolver returns a list of coerced values that comply with the API schema [38].

3.4 Queries and Mutations

Possible GraphQL operations include a *query*, a *mutation* and a *subscription* [39]. The keywords can be used as an operation name in multi-operational applications to specify its purpose for easier debugging. A simple query is a “read” operation that can be compared to a HTTP GET request. As was previously defined, a GraphQL query reflects the shape of its JSON response, fetching certain fields of the objects. The returned response contains all the field types, defined in the schema: meaning that if a field had a type of String, API will return a String. GraphQL queries “can traverse related objects and their fields” [37], making it possible to fetch measurable amounts of related data in one request. There is also an option of passing arguments to any field and nested object in a query, including scalar fields. On the server-side field arguments customize the response to be resolved for the field.

Variables in GraphQL are used to write dynamic arguments to the fields in a query [40]. The static value of the argument in the query is replaced with a dynamic value. The variable is then declared to be accepted by the query and is defined in a separate transport specific dictionary. Using variables is considered a good practice for denoting which arguments in the query are expected to be dynamic. This is done to avoid string interpolation to construct queries from user-supplied values [40].

GraphQL Documentation states that the variable definitions are prefixed by \$, followed by their type. All declared variables must be either scalars, enums, or input object types. So if you want to pass a complex object into a field, you need to know what input type that matches on the server. Variable can be defined as optional or required using a “!” notation next to its type. Variable may have a default value, that is overridden if any variables are passed as part of the variables dictionary. Table 4 shows an example of the use of a variable in a GraphQL query.

TABLE 4. Variable use in a query.

| Query | Response |
|---|---|
| <pre> query HobbitLocationName (\$chapter: Chapter) { location (chapter: \$chapter) { name } } </pre> | <pre> { "data": { "location": { "name": "Hobbiton" } } } </pre> |
| <p>Variable:</p> <pre> { "chapter": "An Unexpected Party" } </pre> | |

Curly braces in the GraphQL query are called selection sets ^[41]. When we request a field that returns an object or an array of objects, a new selection set is necessary to specify which scalar properties are requested, because innermost selection set must contain fields that resolve to scalar values. A field that resolves to an object or an array of objects is also known as a complex field. A complex field requested without a selection set will return GraphQL error. When there is a need to access multiple arguments of a certain field, aliases can be used. Aliases ^[40] rename the results of the field, allowing to avoid conflicts in the response. This use case is illustrated in a table below, where the hero data is not divided on a server side but can be fetched using aliases in a single request.

TABLE 5. Aliases use in the query.

| Query | Response |
|--|--|
| <pre>{ hobbitHero: hero (chapter: "An Unexpected Party") { name } elfHero: hero (chapter: "A Warm Welcome") { name } }</pre> | <pre>{ "data": { "hobbitHero": { "name": "Bilbo" }, "elfHero": { "name": "Legolas" } } }</pre> |

As mentioned by Aleksi Ritsilä in [42], fragmentation is allowed when querying with GraphQL. When comparing two sets of data, reusable fragment units can be defined in queries where they are needed to retrieve isolated data without duplicating the query logic. It is not possible to use fragment on their own, and they must be prefixed with a spread operator to accompany a query.

TABLE 6. Fragmentation in the query.

| Query | Response |
|--|---|
| <pre>{ comparisonLeft: hero (chapter: "An Unexpected Party") { ...comparisonFields } comparisonRight: hero (chapter: "A Warm Welcome") { ...comparisonFields } } fragment comparisonFields on Character{ name origin }</pre> | <pre>{ "data": { "comparisonLeft": { "name": "Bilbo", "age": "111", "origin": "Hobbiton" }, "comparisonRight": { "name": "Legolas", "age": "2931", "origin": "Mirkwood" } } }</pre> |

The response to the GraphQL query can also be modified by the use of directives. A directive can be attached to a field or a fragment. The core GraphQL specification includes two directives: @include and @skip. Based on the value of a Boolean, a field is skipped or included into the query, and therefore the response is modified.

In many cases data modification is a crucial functionality to the API. To modify server-side data GraphQL uses a special type of operation, called a Mutation. A mutation is equivalent of a POST, PUT, DELETE methods in REST, as it can create, update and delete data. Just like in queries, if the mutation field returns an object type, you can ask for nested fields. This can be useful for fetching the new state of an object after an update.

TABLE 7. Mutation and its response.

| Mutation | Response |
|---|---|
| <pre>mutation CreateReviewForEpisode(\$ep: Episode!, \$review: ReviewInput!) { createReview(episode: \$ep, review: \$review) { stars commentary } }</pre> | <pre>{ "data": { "createReview": { "stars": 5, "commentary": "This is a great movie!" } } }</pre> |

3.5 Comparing GraphQL API to REST API

Thingsee Open API has Representational State Transfer or REST architecture. The key abstraction of this architecture is a resource, which is a piece of information such as a document, a temporal service, an image or a collection of other resources, that requires a resource identifier [43]. A resource representation is a state of the resource at a given moment, and it includes a hypermedia link that returns a response when addressed via a request. A request consists of the

endpoint (or the hypermedia URL of the resource that needs to be accessed, where the path determines the resource that is requested), the method (that defines what type of operation will be performed on the data), the request headers and the body of the request ^[44]. The methods refer to possible actions that are requested from the server: Create, Read, Update and Delete. As stated by M. Rouse at [45]:” a RESTful API uses existing HTTP methodologies defined by the RFC 2616 protocol, such as: a POST request to Create that resource, a GET request to retrieve a resource, a PUT request to Update a resource, and a DELETE to remove it”.

To compare RESTful architecture to GraphQL APIs, it is essential to look at the differences when it comes to fetching data, receiving response and creating endpoints for the API. In Thingsee Open API as in any other REST API to request data, a call is made to an endpoint, and the server responds back with the response. As shown in the example from the article by A. Ravichandran [46], the endpoint to retrieve author’s “id” in an example REST API would look like:

```
/author/<id>
```

Another two requests would be needed to get the “posts” and “blog topics” from the same API:

```
/author/<id>/posts  
/author/<id>/topics
```

The scaling of the application often means increased data communication, which leads to the increase in the number of the REST endpoints as well. This problem is referred to as “multiple round trips” and results in slower response time. The need to request data in more than one request is also referred to as “under-fetching”, meaning not enough data is returned from a single request. GraphQL on the other hand, has a single endpoint that all data can be requested from, which reduces the number of endpoints dramatically. The fields in a GraphQL request are communicating with the server declaratively, which also allows to avoid the “over-fetching” of the data. As it was concluded in a study of “Migrating to GraphQL: A Practical Assessment”^[47]: the size of JSON responses returned by the API providers had reduced by 94% with GraphQL, from median 93.5 field long response to 5.5 fields. In terms of bytes, a study measured a reduction in size of the response from 9.8 MB (REST) to 86 KB (GraphQL).

Another study “REST vs GraphQL: A Controlled Experiment”^[48] has conducted experimental research on the amount of time that it takes controlled group of developers to perform similar

tasks using GraphQL and REST. The points in the violin graph in Figure 6 represent the time spent by a participant to conclude a task. As it can be seen, the developers spent on the median nine minutes to implement the REST queries, against six minutes to implement the GraphQL queries. Additionally, a survey was conducted to receive an overview on what opinion the developers have on the technologies, which received answers such as: “for me GraphQL is better than REST, because the query structure allows a better visualization of the query response” [48] and other similar opinions in favor of GraphQL. This data suggests a better and faster development experience in software development projects, as well as improved functionality and response time of the APIs.

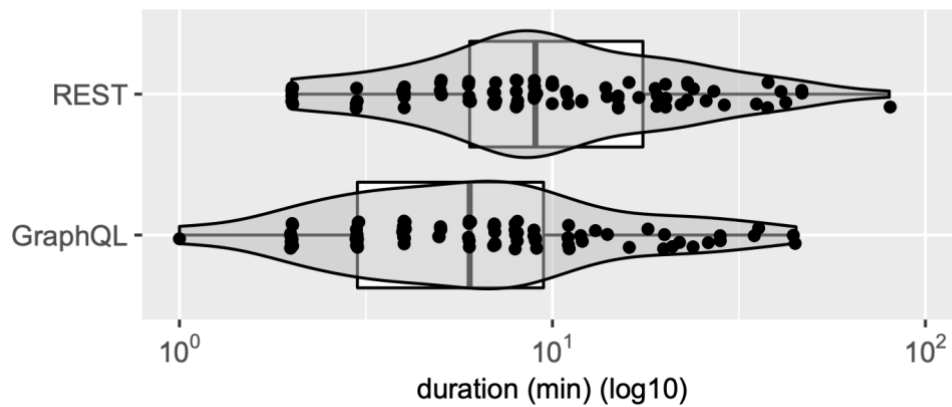


FIGURE 6. Time to conclude the tasks (REST vs GraphQL) in “REST vs GraphQL: A Controlled Experiment” [48].

These factors make GraphQL a feasible tool for development, testing and evaluation of a new experimental version of Thingsee Open API for Haltian Oy, development of which is covered in the next chapter.

4 IMPLEMENTATION (ACTUAL HALTIAN API OPTIMIZATION)

This chapter covers the experimental part of the thesis, including the overview of the structure of the Thingsee Open API, architecture of a new GraphQL API, the development of the solution, deployment to Amazon Web Services and testing with GraphQL Playground.

4.1 Structure of the current API

Chapter 2 covers the technologies that Thingsee Open API utilizes, including Amazon API Gateway service, AWS Lambdas and Amazon Relational Database. The API is built with RESTful architecture, and each resource is accessible through the corresponding endpoint. For example, to get information of a certain sensor or “a thing”, as it is noted in the Thingsee platform, the following endpoint needs to be accessed, where “tuid” is a unique identifier of the sensor and has a type String:

```
GET /things/{tuid}
```

The response includes battery level, identifier of the gateway the sensor sends data to, and private information such as location of the device and its version.

```
{
  "data": {
    "battery_level": 86,
    "timestamp": 1586952604,
    "gateway_tuid": "xxxx00x10733xxxxx", }
}
```

Each sensor has messages, or the raw data values that can be retrieved from the Open API by sending a request to the endpoint, where message id is an identifier that determines what type of sensor messages are requested:

```
GET /things/{tuid}/messages?messageid={messageid}
```

Figure 7 shows all of the endpoints for retrieving sensor information from the Open API. When wanting to retrieve both alerts, commands, group, installations and messages from a sensor, there is a need to make multiple requests to the different endpoints. GET method of each endpoint is triggering a separate Lambda function. Each Lambda function establishes the connection to the Relational Database instance and sends a PostgreSQL query. A response is then returned back in a JSON format. This structure leads to the drawbacks of the RESTful architecture described in earlier chapters: need to fetch from multiple endpoints reveals “under-fetching” and “multiple roundtrips” issues. As not all of the options in the response can be of immediate use in every use case, over-fetching occurs as well.

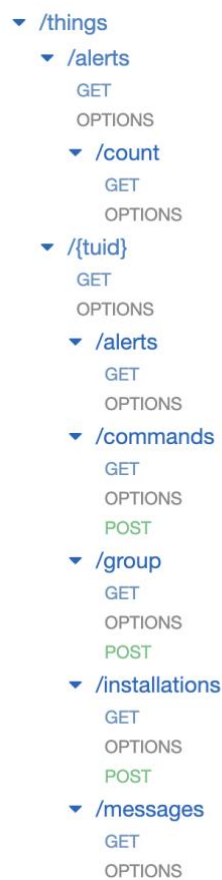


FIGURE 7. Structure of the Thingsee Open API things endpoints.

4.2 Architecture of a new GraphQL solution

The new GraphQL solution will cover all the functionality needed to retrieve the following information from the test database (that is created separately for the solution): all Thingsee devices, a single Thingsee device by tuid and its messages, all Thingsee messages and a group

of Thingsee messages by message id. The project will not cover the retrieval of security sensitive or private fields such as device locations when querying for device information as well as the alerts and versions of the devices. Information such as Thingsee device tuids or unique identifiers will be modified not to reveal the real identifiers of the devices, as well as gateway devices. Thingsee message identifiers will be revealed, as they are specified in the Thingsee public wiki documentation.

Technical requirements include developing the solution that will use the same services as the current Thingsee Open API. As Thingsee platform's Operation Cloud relies strongly on Serverless Framework, compatibility and scalability of the APIs offered in the platform should be maintained. The new GraphQL solution will include an GraphQL Apollo server running on AWS Lambda, that connects to the test Amazon Relational Database, that will include sample data similar to actual database. Running an Apollo server is considered to be one of the best practices to build a production-ready, self-documenting GraphQL API. The API will be accessible in the Amazon API Gateway with a single endpoint.

This solution architecture accumulates scientific and experimental novelty, as at this moment there is a sensible lack of resources that depict the development process of the GraphQL API in combination with Amazon API Gateway, AWS Lambda and Amazon Relational Database service. One of the few approaches put forward by developers to create such a solution is by Luna Davis ^[49], a Senior UI Developer at project202. Her experimental development tutorial allows to create a GraphQL API that retrieves data from a Relational Database instance on MySQL engine. Next section will cover the adaptation of the approach to the Thingsee Open API case that communicates with a PostgreSQL data engine.

4.3 Developing the solution

This section will cover the implementation in detail. Having a working Amazon Web Services account, and AWS Command Line Interface, Node.js and Node Package Manager installed on the development device is a starting requirement.

4.3.1 Creating the Apollo server

Firstly, a new directory was created. In that directory, the following commands were executed:

```
$ npm init
$ npm install --save apollo-server graphql
```

These commands allow to invoke the utility that covers the process of creation of the package.json file in the directory and install Apollo server dependencies. Next, inside of the directory an index.js file is created at ./src/index.js:

```
const { ApolloServer } = require('apollo-server');

const server = new ApolloServer({
  typeDefs,
  resolvers
});

// The 'listen' method launches a web server.
server.listen().then(({ url }) => {
  console.log(`Thingsee server ready at ${url}`);
});
```

The Apollo server can be started by running the command below, and it will be located at the default port 4000. Next, GraphQL schema, resolvers and data sources have to be defined.

```
$ nodemon src/index.js
[nodemon] 2.0.6
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Thingsee server ready at http://localhost:4000/
```

4.3.2 Defining the schema

A schema.js file was created in ./src/schema.js directory. As the functionality of the experimental API will include fetching devices information and messages from two separate PostgreSQL tables in the Amazon Relational Database instance, it is logical to define two GraphQL Object types in the schema:

```
type Thing {
  thingsee_tuid: String!
  message_id: Int!
  gateway_tuid: String!
  battery_level: Int!
  messages: [Message!]!
```

```

}

type Message {
  id: ID!,
  message_id: Int!,
  thingsee_tuid: String!,
  air_pressure: Float!,
  humidity: Float!,
  light: Float!,
  temperature: Float!,
  movecount: Int!,
  distance: Float!,
  timestamp: Int!
}

```

From the above specification, fields and the types are defined on each of the object type. Some fields resolve to a scalar type, such as Int or String, some resolve to another object type. As each Thingsee device contains an array of messages that include raw data, field “messages” on the type Thing resolves to an array of objects, each of those is a Message type. The array is denoted by square brackets.

Both type Thing and type Message have a thingsee_tuid, which is a unique device identifier that is present in both tables in an Amazon Relational Database. It is used as a foreign key that connects the tables. Depending on a thingsee_tuid of the device, messages that have a corresponding thingsee_tuid are fetched. Lastly, a required query type has to be defined to create a single endpoint from which the following queries can be executed: getting things, which returns an array of things, getting a thing by thingsee_tuid, getting messages with a response that contains an array of messages, and getting messages based on their message_id.

```

type Query {
  things: [Thing]!
  thing(thingsee_tuid: String!): Thing
  messages: [Message]!
  message(message_id: Int!): [Message]
}

```

Message identifier allows to filter information based on the type of sensor that sends the messages. In the experimental API, the data is fetched from 3 types of sensors: Thingsee

Environment with message identifier of 12100, Thingsee Presence with an id of 17100 and Thingsee Distance with an id of 13100. Full schema is defined and exported below:

```
const { gql } = require("apollo-server");

const typeDefs = gql`

type Thing {
  thingsee_tuid: String!
  message_id: Int!
  gateway_tuid: String!
  battery_level: Int!
  messages: [Message!]!
}

type Message {
  id: ID!
  message_id: Int!
  thingsee_tuid: String!
  air_pressure: Float!
  humidity: Float!
  light: Float!
  temperature: Float!
  movecount: Int!
  distance: Float!
  timestamp: Int!
}

type Query {
  things: [Thing!]!
  thing(thingsee_tuid: String!): Thing!
  messages: [Message!]!
  message(message_id: Int!): [Message!]!
}

`;

module.exports = typeDefs;
```

4.3.3 Adding data sources

A data source file is created in a `./src/datasources/datasource.js` directory and a following command is executed to install the dependencies:

```
$ npm install --save datasource-sql
```

The data source file includes all the methods that are required to fetch data from the Amazon Relational Database service using “knex”, which is a tool for SQL query building.

```
getDevices() {
  return this.knex.select("*").from("thingsee_devices");
}

getDeviceById(thingsee_tuid) {
  return this.knex
    .select("*")
    .from("thingsee_devices AS thing")
    .leftJoin('messages AS message', 'message.thingsee_tuid', 'thing.thingsee_tuid')
    .where("thing.thingsee_tuid", "=", thingsee_tuid)
    .then((results) => {
      return this.deviceReducer(results);
    });
}

getMessages() {
  return this.knex.select("*").from("messages")
}

getMessageById(message_id) {
  return this.knex
    .select("*")
    .from("messages AS message")
    .where("message.message_id", "=", message_id)
    .options({ nestTables: true })
    .then((results) => {
      return results;
    });
}
```


The above functions send SQL queries to the database tables “thingsee_devices” and “messages”, selecting and joining needed fields and modifying response to fit the schema defining with a reducer function. The reducer function deviceReducer modifies the response from getDeviceById function by mapping all the message fields from an array into a separate array that will be returned for a field messages on a type Thing, as defined in a schema:

```
deviceReducer(devices) {
  if(devices.length > 0) {
    const device = { thingsee_tuid: devices[0].thingsee_tuid, message_id: devices[0].message_id,
gateway_tuid: devices[0].gateway_tuid, battery_level: devices[0].battery_level};
    const messages = devices.map(v => ({ id: v.id, temperature: v.temperature, humidity: v.humidity, light:
v.light, air_pressure: v.air_pressure, }));
    return {
      ...device,
      messages
    };
  }
  return null;
}
```

The data source file is then exported as follows:

```
module.exports = ThingseeDatabase;
```

4.3.4 Adding resolvers

The queries in the schema have to resolve to the functions that send SQL queries to the RDS database tables. Adding a resolver file will make the server functional as the resolvers are responsible for populating the data for every field in a schema. A resolver file was created at ./src/resolver.js directory:

```
module.exports = {
  Query: {
    things: (_, {}, { dataSources }) =>
      dataSources.ThingseeDatabase.getDevices(),
    thing: (_, { thingsee_tuid }, { dataSources }) =>
      dataSources.ThingseeDatabase.getDeviceById(thingsee_tuid),
```

```

messages: (_, {}, { dataSources }) =>
  dataSources.ThingseeDatabase.getMessages(),
message: (_, { message_id }, { dataSources }) =>
  dataSources.ThingseeDatabase.getMessageById(message_id),
}
};

```

As it can be seen, it resolves queries on the schema Query Type to the functions in the data source file as `getDevices ()`, `getDeviceById(thingsee_tuid)`, `getMessages ()`, `getMessageById(message_id)`. Lastly, the schema, resolver and data source file have to be imported into the `index.js` file, as well as Amazon Relational Database credentials to connect to the database instance.

```

const typeDefs = require('./schema');
const ThingseeDatabase = require("./datasources/muscle");
const resolvers = require('./resolvers');
const knexConfig = {
  client: 'postgresql',
  connection: {
    host : 'database.XXX.XXX.rds.amazonaws.com',
    user : 'XXXXXX',
    password : 'XXXXXXXX',
    database : 'database'
  }
};

```

At this point, it is possible to send queries on on the GraphQL Playground at <http://localhost:4000>. Next step is deploying the server to AWS Lambda using Serverless template.

4.4 Deployment

To deploy the server to AWS Lambda it is important to ensure successful configuration of AWS Command line interface as well as installation of the Serverless framework:

```
$ npm install serverless
```

At the root of the directory a `serverless.yml` file is created with the following content:

```
# serverless.yml
```

```

service: GraphQL-API
provider:
  name: aws
  runtime: nodejs12.x
  region: region
  profile: profile
functions:
  graphql:
    # this is formatted as <FILENAME>.<HANDLER>
    handler: src/graphql.graphqlHandler
    events:
      - http:
          path: graphql
          method: post
          cors: true
      - http:
          path: graphql
          method: get
          cors: true

```

A graphHandler is referring to graphql.js file, that should be created in the src folder:

```

const { ApolloServer } = require('apollo-server-lambda');
const typeDefs = require('./schema');
const thingseeDatabase = require("./datasources/muscle");
const resolvers = require('./resolvers');
const knexConfig = {
  client: 'postgresql',
  connection: {
    host : 'database.XXXXXX.XXXXXXX.rds.amazonaws.com',
    user : 'XXXXXXXX',
    password : 'XXXXXXXX',
    database : 'database'
  }
};
const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources: () => ({
    thingseeDatabase: new thingseeDatabase(knexConfig)
  })
});

```

```

    }},
    playground: {
      endpoint: "/dev/graphql"
    }
  });
exports.graphqlHandler = server.createHandler({
  cors: {
    origin: '*',
    credentials: false,
  },
});

```

This file is exported as a graphqlHandler and will create an Apollo server on an AWS Lambda after the deployment. To start the deployment the following script should be executed:

```

$ serverless deploy
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service service-name.zip file to S3 (9.25
MB)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.....
Serverless: Stack update finished...
Service Information
service: service
stage: dev
region: region
stack: stack_name
resources: 13
api keys:
  None
endpoints:
  POST - url for post
  GET - url for get
functions:
  graphql: name_of_the_lambda
layers:
  None

```

```
Serverless: Removing old service artifacts from S3...
Serverless: Run the "serverless" command to setup monitoring,
troubleshooting and testing.
```

There is a new API visible in the API Gateway services. Figure 8 shows a single endpoint on our API. A Lambda function is attached to the methods and will be executed in a testing phase that will be covered in the next section.

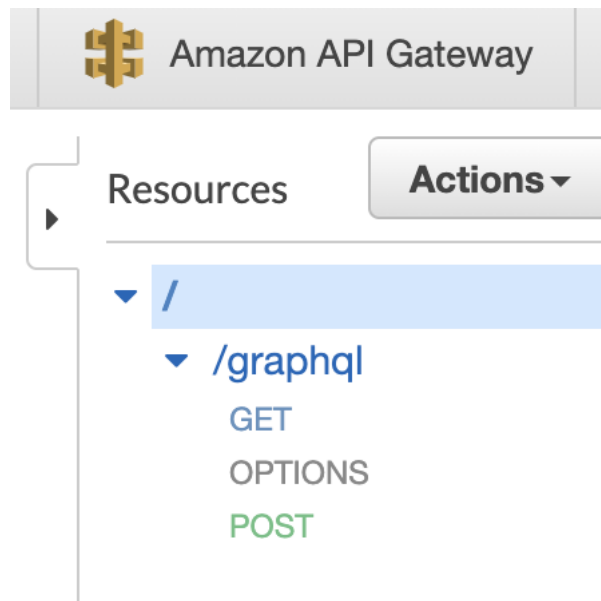
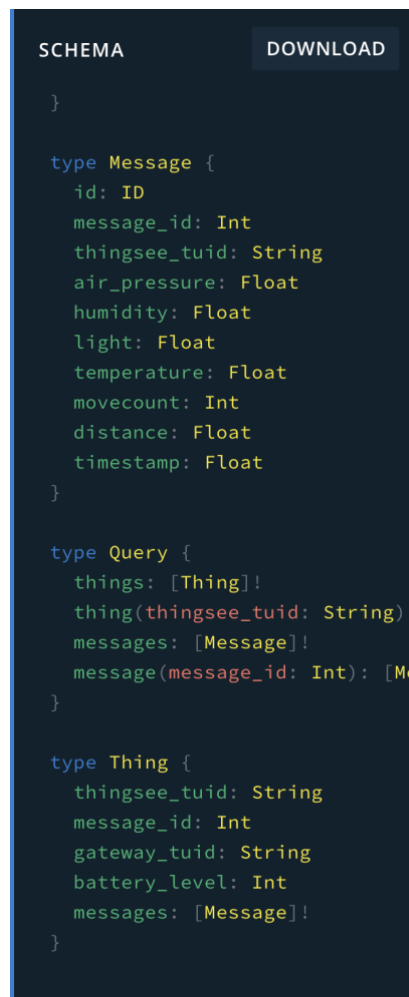


FIGURE 8. Thingsee GraphQL API endpoints.

4.5 Testing

After the successful deployment, two API endpoints were returned: GET and POST. The functionality of the experimental API covers queries (read operations), but not mutations (write operations). Therefore, an URL returned that is marked for a GET request needs to be opened. A URL opened in browser shows a graphical interface that is suitable for the testing of the API called GraphQL Playground. It is a graphical, interactive, in-browser GraphQL IDE, created by Prisma and based on GraphiQL. As explained in Apollo documentation: during the development stage, Apollo Server enables GraphQL Playground on the same URL as the GraphQL server itself and automatically serves the GUI to web browsers^[50]. When the project is set to production, GraphQL Playground is disabled as a production best-practice. The schema defined for the API is fetched automatically and can be accessed through the “schema” option on the right side of the

interface. As shown on Figure 9, the schema contains types that were defined in a previous section.



The image shows a screenshot of the GraphQL Playground interface. At the top, there are two tabs: 'SCHEMA' and 'DOWNLOAD'. The 'SCHEMA' tab is active, displaying a GraphQL schema definition. The schema defines three types: Message, Query, and Thing. The Message type has fields: id (ID), message_id (Int), thingsee_tuid (String), air_pressure (Float), humidity (Float), light (Float), temperature (Float), movecount (Int), distance (Float), and timestamp (Float). The Query type has fields: things ([Thing]!), thing(thingsee_tuid: String!), messages ([Message]!), and message(message_id: Int!) (partially visible). The Thing type has fields: thingsee_tuid (String), message_id (Int), gateway_tuid (String), battery_level (Int), and messages ([Message]!).

```

}

type Message {
  id: ID
  message_id: Int
  thingsee_tuid: String
  air_pressure: Float
  humidity: Float
  light: Float
  temperature: Float
  movecount: Int
  distance: Float
  timestamp: Float
}

type Query {
  things: [Thing]!
  thing(thingsee_tuid: String!): Thing!
  messages: [Message]!
  message(message_id: Int!): [Message]!
}

type Thing {
  thingsee_tuid: String
  message_id: Int
  gateway_tuid: String
  battery_level: Int
  messages: [Message]!
}
```

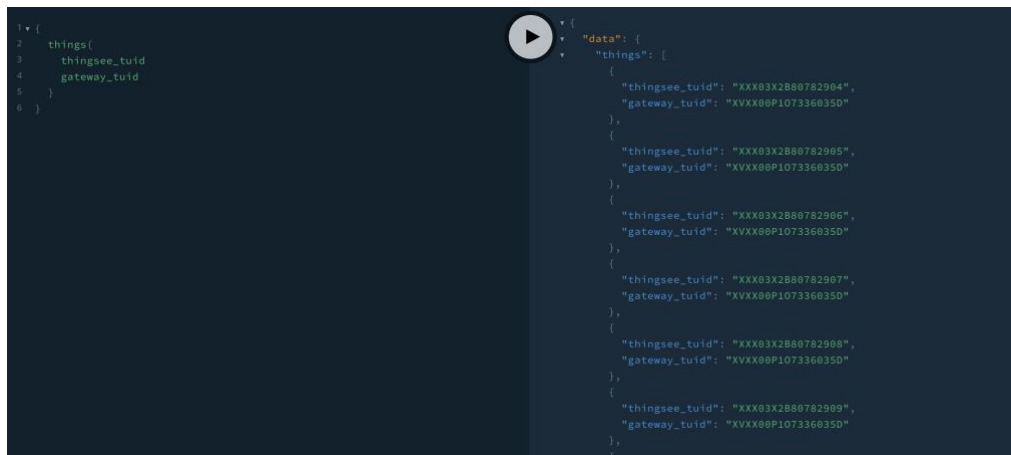
FIGURE 9. Schema in GraphQL Playground.

There are four types of queries that can be sent to the experimental API: getting all device data (things), getting a device information and messages by device tuid, getting all message data (messages), and lastly, getting messages based on the message_id. The chapter will proceed with the testing of each of the queries in the GraphQL Playground.

4.5.1 Get all devices

A type of Thing has the following fields that can be requested in a query: thingsee_tuid: String, message_id: Int, gateway_tuid: String and battery_level: Int. It is possible to ask for as many fields at once as needed. A query is constructed on the left side of the graphical interface. Note

that the query “things” has two selection sets or sets of curly braces. The response shown in Figure 10 is in JSON format and reflects exactly the fields on the object type that were requested.



```

1 {
2   things(
3     thingsee_tuid
4     gateway_tuid
5   )
6 }

```

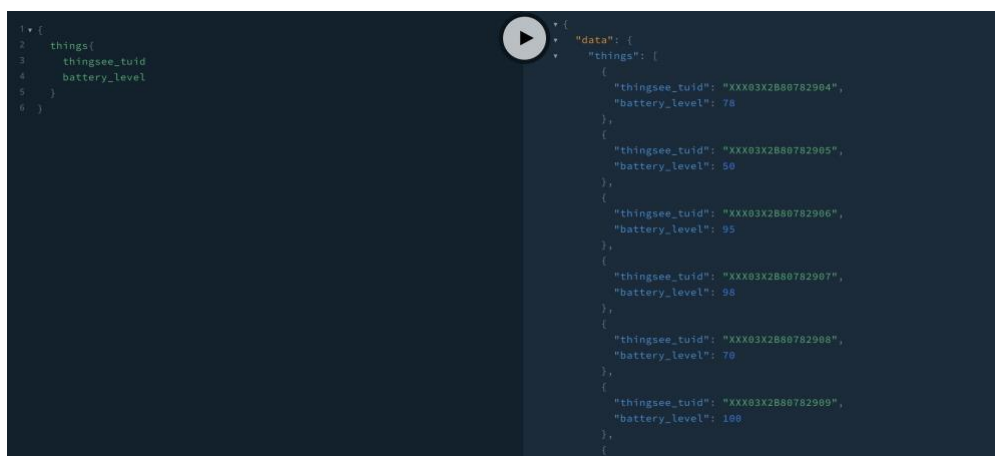
```

{
  "data": {
    "things": [
      {
        "thingsee_tuid": "XXX03X2880782904",
        "gateway_tuid": "XVXX00P107336035D"
      },
      {
        "thingsee_tuid": "XXX03X2880782905",
        "gateway_tuid": "XVXX00P107336035D"
      },
      {
        "thingsee_tuid": "XXX03X2880782906",
        "gateway_tuid": "XVXX00P107336035D"
      },
      {
        "thingsee_tuid": "XXX03X2880782907",
        "gateway_tuid": "XVXX00P107336035D"
      },
      {
        "thingsee_tuid": "XXX03X2880782908",
        "gateway_tuid": "XVXX00P107336035D"
      },
      {
        "thingsee_tuid": "XXX03X2880782909",
        "gateway_tuid": "XVXX00P107336035D"
      }
    ]
  }
}

```

FIGURE 10. Things (tuid, gateway tuid) in GraphQL Playground.

Same query can be modified to request battery_level instead of gateway_tuid. The response replaces that field, without “over-fetching”, or including the fields on the object that were not specified in the query as shown in Figure 11.



```

1 {
2   things(
3     thingsee_tuid
4     battery_level
5   )
6 }

```

```

{
  "data": {
    "things": [
      {
        "thingsee_tuid": "XXX03X2880782904",
        "battery_level": 78
      },
      {
        "thingsee_tuid": "XXX03X2880782905",
        "battery_level": 90
      },
      {
        "thingsee_tuid": "XXX03X2880782906",
        "battery_level": 95
      },
      {
        "thingsee_tuid": "XXX03X2880782907",
        "battery_level": 98
      },
      {
        "thingsee_tuid": "XXX03X2880782908",
        "battery_level": 70
      },
      {
        "thingsee_tuid": "XXX03X2880782909",
        "battery_level": 100
      }
    ]
  }
}

```

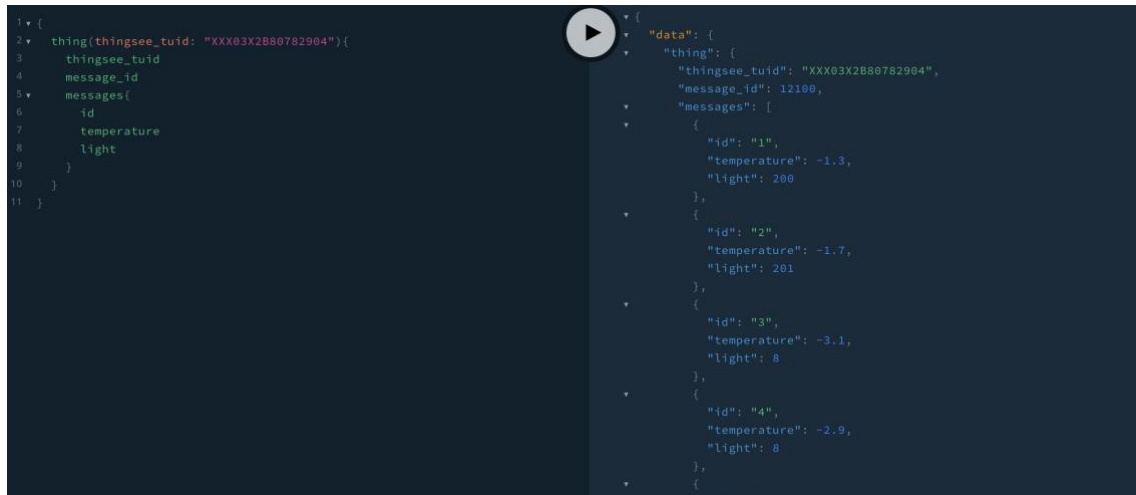
FIGURE 11. Things (gateway tuid, battery level) in the GraphQL Playground.

It is safe to conclude that the query for getting all devices is executed without any issues. The response is expectable, the connection to the database is successful.

4.5.2 Get device and its messages by device identifier

Next query shown in Figure 12 accepts an argument of “thingsee_tuid” of type String. Before the new selection set the argument is passed in a set of parenthesis similarly to a JavaScript

function: thing(thingsee_tuid: "XXX03X2B80782904"). As the query expects to receive a string argument, the value of "thingsee_tuid" should come in quotation marks to specify its type. As each individual sensor has a set of messages, to retrieve those a new selection set is opened. There, fields of type "Message" are specified.



```

1 {
2   thing(thingsee_tuid: "XXX03X2B80782904"){
3     thingsee_tuid
4     message_id
5     messages{
6       id
7       temperature
8       light
9     }
10  }
11 }

```

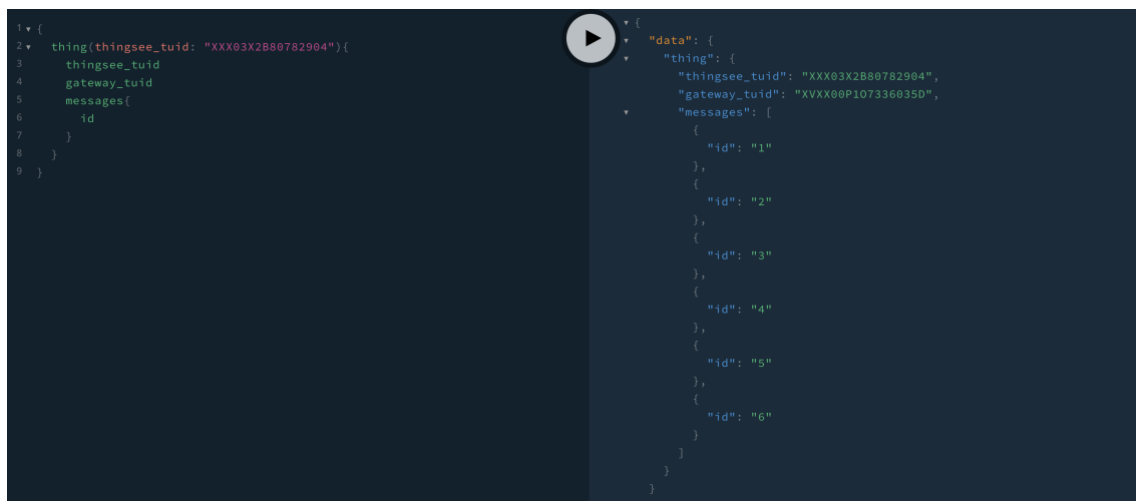
```

{
  "data": {
    "thing": {
      "thingsee_tuid": "XXX03X2B80782904",
      "message_id": 12100,
      "messages": [
        {
          "id": "1",
          "temperature": -1.3,
          "light": 200
        },
        {
          "id": "2",
          "temperature": -1.7,
          "light": 201
        },
        {
          "id": "3",
          "temperature": -3.1,
          "light": 8
        },
        {
          "id": "4",
          "temperature": -2.9,
          "light": 8
        }
      ]
    }
  }
}

```

FIGURE 12. Get thing by tuid in the GraphQL Playground.

In the request at Figure 12, all messages for a device with "thingsee_tuid" of "XXX03X2B80782904" are returned, with the fields of "id", "temperature" and "light" selected in a query. The response includes only the data that is declared in a request, just as in the previous case. The query successfully maps messages that have a "thingsee_tuid" equal to "XXX03X2B80782904" from a separate table in the Amazon relational database, and the reducer pushes that data into the array that is returned for the field "messages". The request in Figure 12 illustrates a subsequent query, that asks for the different fields on the object type "Thing".



```

1 {
2   thing(thingsee_tuid: "XXX03X2B80782904"){
3     thingsee_tuid
4     gateway_tuid
5     messages{
6       id
7     }
8   }
9 }

```

```

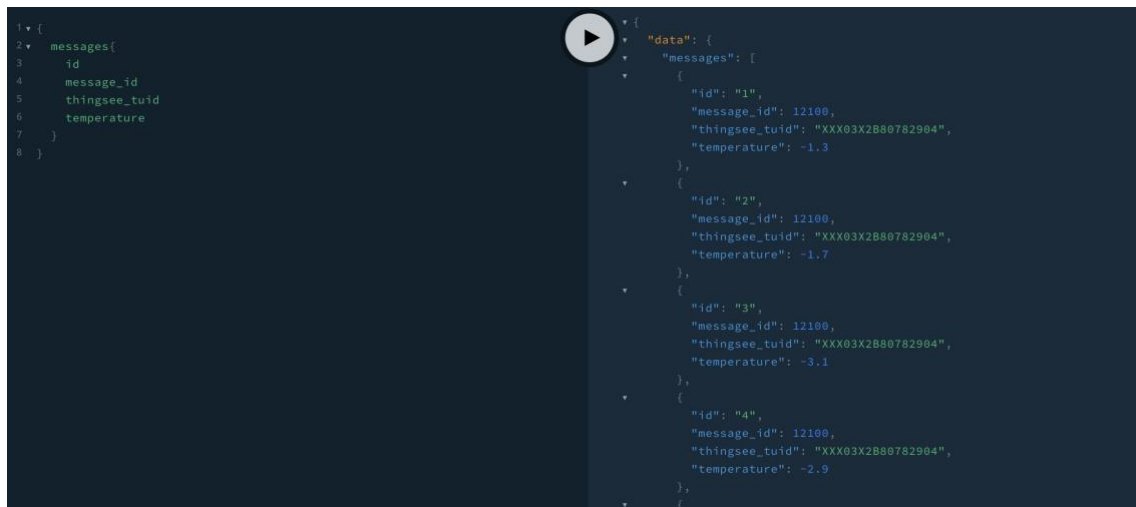
{
  "data": {
    "thing": {
      "thingsee_tuid": "XXX03X2B80782904",
      "gateway_tuid": "XVXX00P107336035D",
      "messages": [
        {
          "id": "1"
        },
        {
          "id": "2"
        },
        {
          "id": "3"
        },
        {
          "id": "4"
        },
        {
          "id": "5"
        },
        {
          "id": "6"
        }
      ]
    }
  }
}

```

FIGURE 13. Subsequent fields on a "Thing" in the GraphQL Playground

4.5.3 Get all messages

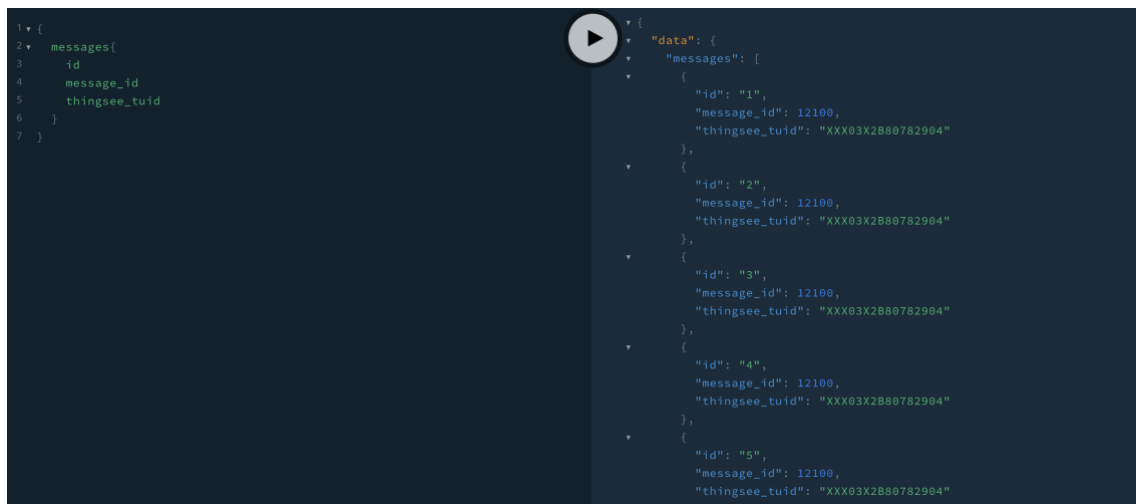
Type “Message” has two following fields that can be requested in a query: an “id” that increments, “message_id” of scalar type Int, “thingsee_tuid” of type String and sensor specific fields such as “air_pressure”, “humidity”, “light”, “temperature”, “movecount”, “distance”. The response is successful as shown in Figure 14 and Figure 15, just as in previous test scenarios, the data is fetched from the “messages” table in the database service and is returned as specified in a query. Next query tests for a different field on the same object type of “Message”.



```
1 {
2   messages {
3     id
4     message_id
5     thingsee_tuid
6     temperature
7   }
8 }
```

```
{
  "data": {
    "messages": [
      {
        "id": "1",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "temperature": -1.3
      },
      {
        "id": "2",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "temperature": -1.7
      },
      {
        "id": "3",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "temperature": -3.1
      },
      {
        "id": "4",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "temperature": -2.9
      }
    ]
  }
}
```

FIGURE 14. Messages in the GraphQL Playground



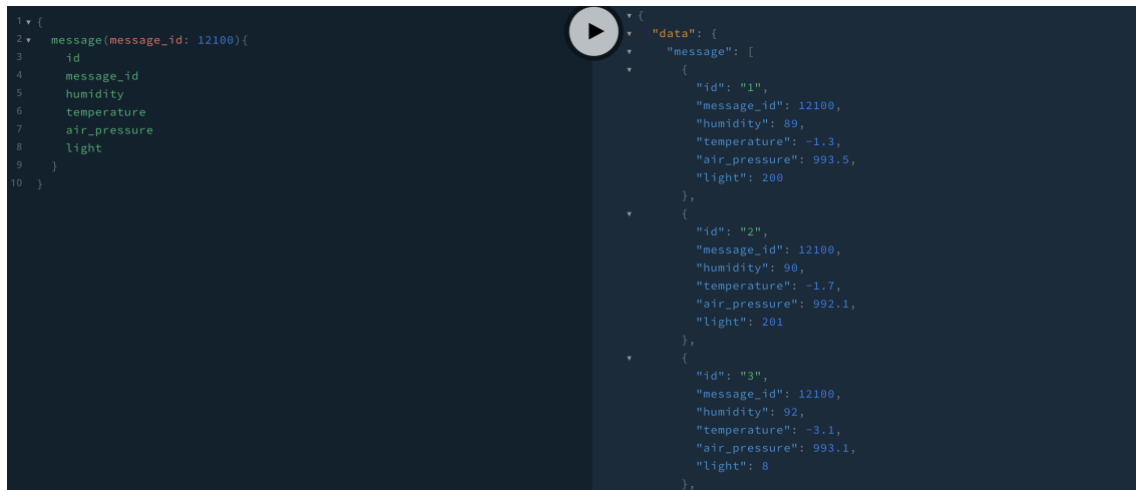
```
1 {
2   messages {
3     id
4     message_id
5     thingsee_tuid
6   }
7 }
```

```
{
  "data": {
    "messages": [
      {
        "id": "1",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904"
      },
      {
        "id": "2",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904"
      },
      {
        "id": "3",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904"
      },
      {
        "id": "4",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904"
      },
      {
        "id": "5",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904"
      }
    ]
  }
}
```

FIGURE 15. Subsequent fields on Message in the GraphQL Playground

4.5.4 Get messages by message identifier

Similarly to querying for device information by passing an id as an argument, in this query shown in Figure 16 the “message_id” is passed. Message_id of 12100 is for the Thingsee Environment sensor, so it is possible to request raw data fields as “temperature”, “humidity”, “air_pressure” and “light”.



```
1 {
2   message(message_id: 12100){
3     id
4     message_id
5     humidity
6     temperature
7     air_pressure
8     light
9   }
10 }
```

```
{
  "data": {
    "message": [
      {
        "id": "1",
        "message_id": 12100,
        "humidity": 89,
        "temperature": -1.3,
        "air_pressure": 993.5,
        "light": 200
      },
      {
        "id": "2",
        "message_id": 12100,
        "humidity": 90,
        "temperature": -1.7,
        "air_pressure": 992.1,
        "light": 201
      },
      {
        "id": "3",
        "message_id": 12100,
        "humidity": 92,
        "temperature": -3.1,
        "air_pressure": 993.1,
        "light": 8
      }
    ]
  }
}
```

FIGURE 16. Message by message id for Thingsee Environment in the GraphQL Playground

Next, testing same query with different selected fields. Figure 17 shows that not only the response mirrors the query, but the values for each field are also complacent with the typing system. The types of the values that are coming from the database are the same as was identified in the schema. Figure 18 shows query and response for Thingsee Presence sensor with an identifier of 17100, and Figure 19 for Thingsee Distance sensor with a message identifier equal to 13100.

```

1 {
2   message(message_id: 12100){
3     id
4     message_id
5     thingsee_tuid
6     humidity
7     temperature
8   }
9 }

```

```

{
  "data": {
    "message": [
      {
        "id": "1",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "humidity": 89,
        "temperature": -1.3
      },
      {
        "id": "2",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "humidity": 90,
        "temperature": -1.7
      },
      {
        "id": "3",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "humidity": 92,
        "temperature": -3.1
      },
      {
        "id": "4",
        "message_id": 12100,
        "thingsee_tuid": "XXX03X2B80782904",
        "humidity": 93,
        "temperature": -3.5
      }
    ]
  }
}

```

FIGURE 17. Response for Thingsee Environment in the GraphQL Playground

```

1 {
2   message(message_id: 17100){
3     id
4     message_id
5     movecount
6   }
7 }

```

```

{
  "data": {
    "message": [
      {
        "id": "7",
        "message_id": 17100,
        "movecount": 2
      },
      {
        "id": "8",
        "message_id": 17100,
        "movecount": 2
      },
      {
        "id": "9",
        "message_id": 17100,
        "movecount": 0
      },
      {
        "id": "10",
        "message_id": 17100,
        "movecount": 9
      }
    ]
  }
}

```

FIGURE 18. Response for Thingsee Presence in the GraphQL Playground

```

1 {
2   message(message_id: 13100){
3     id
4     message_id
5     thingsee_tuid
6     distance
7   }
8 }

```

```

{
  "data": {
    "message": [
      {
        "id": "11",
        "message_id": 13100,
        "thingsee_tuid": "XXX06E2P83602011",
        "distance": 20
      },
      {
        "id": "12",
        "message_id": 13100,
        "thingsee_tuid": "XXX06E2P83602012",
        "distance": 19.5
      },
      {
        "id": "13",
        "message_id": 13100,
        "thingsee_tuid": "XXX06E2P83602013",
        "distance": 19
      },
      {
        "id": "14",
        "message_id": 13100,
        "thingsee_tuid": "XXX06E2P83602014",
        "distance": 18.5
      }
    ]
  }
}

```

FIGURE 19. Response for Thingsee Distance in the GraphQL Playground

5 EVALUATION

The experimental Thingsee Open GraphQL API has applied existing technologies such as Amazon API Gateway, AWS Lambda and Amazon Relational Database successfully. The queries fetch the data from the database instance without delay, keeping the SQL queries shorter as the response can be modified from a query, rather than on the server-side. The number of endpoints for the same functionality has decreased from 7 endpoints to a single endpoint. With more functionality added to the API the number of endpoints in the Thingsee GraphQL API will still be equal to one. In prospect, if all 22 endpoints are replaced with one smart GraphQL endpoint, the overall reduction of the number of the endpoints will be around 95.5%.

The compliance with the existing technologies allows for possibilities of further development of the GraphQL API, as other Hiltian developers have expertise with the Serverless framework and Amazon Relational Database in PostgreSQL engine. As the GraphQL queries and mutations relieve API architecture of the complexity, it is important to test the approach with a database structure similar in complexity to the actual database used by Thingsee Open API. That will allow for more accurate evaluation of the capabilities of the GraphQL on production level, dealing with more complex queries and larger amounts of data.

The amount of data returned by the queries is highly customizable, as the testing phase has shown. That can result in less custom data integrations both on the server side and in the Thingsee Operations Cloud, resulting in more agile development process and faster project delivery times. The structure and the typing system of the API has sparked interest as a favorable practice for maintenance and error handling in projects that involve high number of sensors, gateways and data communication.

Another advantage of the solution is the easier versioning of the new GraphQL API, that will maintain the endpoint for the customers. It is important to consider limitations in caching and complexity with high level error handling that may occur in big scale production stage API. Nevertheless, further work on the API was approved. When the GraphQL API will cover most of the Open API functionality, it can be tested with the new or existing customer integrations to test the potentials of the GraphQL in the context of Internet of Things further.

6 CONCLUSION

The main goal of the research was to build a strong theoretic knowledge base, a thorough understanding of the GraphQL query language and the principles in its core. The confidence in the understanding of schema, types, resolvers as well as current technologies of the Thingsee Open API as Amazon API Gateway, AWS Lambda and Relational Database service allowed for successful development of a solution that brings a novel value to the world of Internet of Things. Among a very few approaches to the solution, it was challenging to navigate to a ready implementation that covered all planned functionality and has proven to be a hopeful resolution to the problems of versioning, over and under fetching, growing number of endpoints with the existing Open API.

A thorough analysis of more than 50 resources on the topic allowed to grasp the potential of the new growing technology in a rapidly evolving field of the Internet of Things, where data communication comes first. The need to constantly evolve and develop new approaches to solve existing problems in web development is not only directed at improving the performance of the applications, but also allows for a better development experience for the software developers. Software development shows tendencies of shifting the focus from the endless increase in production value to the betterment of day-to-day web development of new solutions, long-term maintenance and security. The Serverless framework aims to redirect attention and effort to the functionality that brings value, instead of accumulating more effort to solve and maintain tedious server-side challenges. In that sense, GraphQL is aiming to do the same. As the focus is shifted away from dealing with the drawbacks of the RESTful architecture, the world of web development can focus on more innovative application logic, frontend development in various fields from eCommerce to Medical tech to the Internet of Things. Novelty always presumes new challenges, and those will have to be resolved at time as well, completing a circle of “life” of the technology, opening the door to a better and more innovative approach.

REFERENCES

- [1] S. Ranger, "The Internet of Things explained. What the IoT is, and where it's going next" Part of a ZDNET special feature: 5G: What it means for IoT, February 3, 2020. [Online], Available: <https://www.zdnet.com/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now/>. [Accessed Sept. 4, 2020].
- [2] T. Reenskaug, J. O. Coplien, "The DCI Architecture: A New Vision of Object-Oriented Programming" Artima Developer, March 20, 2009. [Online], Available: https://web.archive.org/web/20090323032904/https://www.artima.com/articles/dci_vision.html. [Accessed Sept. 21, 2020].
- [3] Cake Software Foundation, "Cake PHP Overview: Understanding Model-View-Controller" 2020. [Online]. Available: <https://book.cakephp.org/2/en/cakephp-overview/understanding-model-view-controller.html>. [Accessed Sept. 1, 2020].
- [4] Developer Chrome, "MVC Architecture". [Web blog post]. Available: https://developer.chrome.com/apps/app_frameworks. [Accessed Sept. 1, 2020]
- [5] Dipen Patel, "An Introduction to MVC Architecture: A Web Developer's Point of view" Web Dev Zone, July 10, 2019. [Web Article]. Available: <https://dzone.com/articles/introduction-to-mvc-architecture-web-developer-poi>. [Accessed Sept. 2, 2020]
- [6] Samer Buna, Learning GraphQL and Relay. Birmingham, UK: Packt Publishing Ltd, 2016. [Book]
- [7] Nick Schrock, "GraphQL Introduction" React, May 1, 2015. [Web article]. Available: <https://reactjs.org/blog/2015/05/01/graphql-introduction.html>. [Accessed Sept. 5, 2020]
- [8] Stackshare, "Who uses GraphQL". [Web blog post]. Available: <https://stackshare.io/graphql> [Accessed Sept. 6, 2020]

- [9] Haltian, "IoT Services for Businesses" haltian.com, para. 2, 2020. [Online]. Available: <https://haltian.com/service/iot/>. [Accessed Oct. 4, 2020].
- [10] Haltian, "World class plug-and-play IoT hardware and software products". [Online]. Available: <https://haltian.com/products/> [Accessed Oct. 7, 2020]
- [11] Thingsee Wiki, "Thingsee Sensors". [Technical documentation]. Available: <http://139.59.148.143/platform/sensors> [Accessed Oct. 10, 2020]
- [12] Wirepas, "Our product – Wirepas Mesh". [Technical documentation]. Available: <https://wirepas.com/what-is-wirepas-mesh/> [Accessed Oct. 10, 2020]
- [13] Open Automation Software, "What is an IoT Gateway?". [Online]. Available: <https://openautomationsoftware.com/open-automation-systems-blog/what-is-an-iotgateway/#:~:text=An%20IoT%20Gateway%20is%20a,software%20that%20performs%20essential%20tasks> [Accessed Oct. 11, 2020]
- [14] Thingsee Wiki, "Thingsee Gateways". [Technical Documentation]. Available: http://139.59.148.143/platform/iot_devices [Accessed Oct. 11, 2020]
- [15] Thingsee Wiki, "Thingsee Operations Cloud". [Technical Documentation]. Available: <http://139.59.148.143/platform/operations> [Accessed Oct. 12, 2020]
- [16] Thingsee Wiki, "Thingsee IoT as a Service". [Image]. Available: <http://139.59.148.143/start> [Accessed Oct. 13, 2020]
- [17] Haltian Support, "About Thingsee Services API". [Web blog post]. Available: <https://support.haltian.com/api/open-services-api/> [Accessed Oct. 14, 2020]
- [18] Serverless framework, "AWS Introduction". [Technical Documentation]. Available: <https://www.serverless.com/framework/docs/providers/aws/guide/intro/> [Accessed Oct. 13, 2020]

- [19] Serverless, “AWS Lambda, The Ultimate Guide”. [Online Technical Guide]. Available: <https://www.serverless.com/aws-lambda> [Accessed Oct. 14, 2020]
- [20] AWS Serverless, Serverless Computing, “Serverless on AWS”. [Web blog post]. Available: <https://aws.amazon.com/serverless/> [Accessed Oct. 14, 2020]
- [21] AWS Lambda, “AWS Lambda, Benefits”. [Web blog post]. Available: <https://aws.amazon.com/lambda/?c=ser&sec=srv> [Accessed Oct. 14, 2020]
- [22] Serverless, “Why Serverless?”. [Online Technical Guide]. Available: <https://www.serverless.com/learn/why/> [Accessed Oct. 14, 2020]
- [23] Amazon API Gateway, “Overview”. [Online Technical documentation]. Available: <https://aws.amazon.com/api-gateway/> [Accessed Oct. 14, 2020]
- [24] B. McNamara, J. Pirtle, T. Bruce for AWS Whitepaper, “Security Overview of Amazon API Gateway”. Page 1, “About Amazon Gateway”, Paragraph 2. [Technical Report]. Available: https://d1.awsstatic.com/whitepapers/api-gateway-security.pdf?svrd_sip6 [Accessed Oct. 14, 2020]
- [25] Developer’s Guide, “What is Amazon API Gateway?”. [Image]. Available: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html> [Accessed Oct. 13, 2020]
- [26] Serverless, “AWS Lambda, The Ultimate Guide”. [Online Technical Guide]. Available: <https://www.serverless.com/aws-lambda> [Accessed Oct. 14, 2020]
- [27] AWS Lambda, “How it works”. [Image]. Available: <https://aws.amazon.com/lambda/> . [Accessed Oct. 13, 2020]
- [28] Amazon RDS, “Amazon relational database Service (RDS)”. [Online Technical Guide]. Available: <https://aws.amazon.com/rds/lambda> [Accessed Oct. 21, 2020]

- [29] PostgreSQL Tutorial, “What is PostgreSQL?” [Tutorial]. Available: <https://www.postgresqltutorial.com/what-is-postgresql/> [Accessed Oct. 21, 2020]
- [30] Technopedia, “Query Language” [Dictionary]. Available: <https://www.techopedia.com/definition/3948/query-language> [Accessed Oct. 25, 2020]
- [31] G. Brito, T. Mombach, M. Tulio Valente, “Migrating to GraphQL: A Practical Assessment”, ASERG Group, Department of Computer Science, Federal University of Minas Gerais, Brazil. Available: <https://arxiv.org/pdf/1906.07535.pdf> [Accessed Oct. 27, 2020]
- [32] GraphQL.org “A query language for your API”, para. 2, 2020. [Online]. Available: <https://graphql.org/>. [Accessed Oct. 27, 2020].
- [33] Buna, S. 2016 Packt Publishings. Learning GraphQL and Relay. [Book]. Birmingham, UK. Available: https://books.google.fi/books?hl=en&lr=&id=j6XWDQAAQBAJ&oi=fnd&pg=PP1&dq=graphql+internet+of+things&ots=CQGA4wVKd4&sig=wn7QLCKtOxe_tXhg5B2-CnJz9qM&redir_esc=y#v=onepage&q&f=false
- [34] How to GraphQL, “GraphQL is the better REST”, [Tutorial]. Available: <https://www.howtographql.com/basics/1-graphql-is-the-better-rest/> [Accessed Oct. 21, 2020]
- [35] O. Hartig and J. Perez, “An initial analysis of Facebook’s GraphQL language,” in 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web (AMW), 2017, pp. 1–10.
- [36] G. Mirza, “GraphQL Schema Definition” EdnSquare, [Online]. Available: <https://ednsquare.com/story/graphql-sdl-introduction-to-schema-definition-language> [Accessed Oct. 21, 2020]
- [37] GraphQL, “Queries and Mutations”. [Online Technical documentation]. Available: <https://graphql.org/learn/queries/> [Accessed Oct. 29, 2020]

- [38] GraphQL, “Root fields & resolvers”. [Online Technical documentation]. Available: <https://graphql.org/learn/execution/> [Accessed Oct. 29, 2020]
- [39] C. Nwamba, “Your First Look at GraphQL Queries, Mutations, and Subscriptions” Developer central, JavaScript. June 19, 2019. [Online] Available: <https://www.telerik.com/blogs/your-first-look-at-graphql-queries-mutations-and-subscriptions> [Accessed Oct. 29, 2020]
- [40] GraphQL, “Variables”. [Online Technical documentation]. Available: <https://graphql.org/learn/queries/> [Accessed Oct. 29, 2020]
- [41] A. Aiyer, “GraphQL Selection Sets” Medium.com, May 26, 2016. [Online Article]. Available: <https://medium.com/front-end-developers/graphql-selection-sets-d588f6782e90> [Accessed Nov. 3, 2020]
- [42] A. Ritsilä, “GraphQL: The API Design Revolution” B. S. Thesis, Haaga-Helia University of Applied Sciences, 2017.
- [43] REST API Tutorial, “What is REST?” [Online]. Available: <https://restfulapi.net/> Accessed Nov. 3, 2020]
- [44] Zell, “Understanding and Using REST APIs” SmashingMagazine.com, January 17, 2018. [Online Article]. Available: <https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/> [Accessed Nov. 9, 2020]
- [45] M. Rouse, “RESTful API (REST API)” TechTarget, September 2020. [Online Article]. Available: <https://searcharchitecture.techtarget.com/definition/RESTful-API> [Accessed Nov. 10, 2020]
- [46] A. Ravichadran, “GraphQL vs. REST- A Comparison” Medium.com, September 23, 2019. [Online Article]. Available: <https://adhithiravi.medium.com/graphql-vs-rest-a-comparison-16a2f5f29198> [Accessed Nov. 10, 2020]

- [47] G. Brito, M. T. Valente, “Migrating to GraphQL: A Practical Assessment” Department of Computer Science, Federal University of Minas Gerais, Brazil. Available: <https://arxiv.org/pdf/1906.07535.pdf> [Accessed Nov. 10, 2020]

- [48] G. Brito, M. T. Valente “REST vs GraphQL: A Controlled Experiment” Department of Computer Science (DCC), Federal University of Minas Gerais, Brazil. Available: <https://arxiv.org/pdf/2003.04761.pdf> [Accessed Nov. 20, 2020]

- [49] L. Davis, ““How to Create an End-to-End Serverless App with React, Apollo, Lambda and AWS RDS” project 202, [Online]. Available: <https://www.projekt202.com/blog/2020/create-an-end-to-end-serverless-app> [Accessed Nov. 29, 2020]