Efficient data communication between a webclient and a cloud environment

Kit Gustavsson
kitgustavsson@gmail.com
Erik Stenlund
erikstenlund0810@gmail.com

Axis Communications
Emdalavägen 14

Advisor: Maria Kihl

June 23, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

Modern single page web applications that continuously transfer a lot of data between the clients and the server require new techniques for communicating this data. The REST architecture has for a long time been the most common solution when developing web APIs but the technique GraphQL has in recent time become an interesting alternative to REST.

This thesis had two major purposes, the first was to investigate and point out the difference between using REST and a GraphQL solution when developing a web API. The second was to, based on the results of the investigation, create a decision model that may be used as support when deciding on what type of technique to use and the effect of the decision when developing a web application. Prototypes of each API were implemented and used to conduct measurements of each technique's performance. Additional infrastructure such as web servers and load generators were also developed for the measurements. The main work behind constructing the decision model was to choose which architectural decisions that are relevant when comparing the two different architectures. This was done by literature studies, interviews with developers and experiences from developing the prototype-APIs. The results from the measurements resulted in a relative performance comparison between the two different techniques. The decision model was also designed and the discussion about it shows the differences between the techniques learned from experience and the studies of them.

Popular scientific summary

If you have ever developed an API, or more specifically a web API you have probably encountered the challenge of deciding which techniques, language and architecture to use. In this thesis we have evaluated and compared REST and GraphQL, two technologies that are commonly used, in terms of performance, maintainability and usability.

The goal was to summarize the comparison both in a decision model covering the topics about maintainability and usability and by producing measurements of the performance. The decision model is also supposed to give aid when trying to decide which technique that is most suitable for an API. It may also shine light on what challenges might be faced later.

Much of the information on the subject today is either about older techniques or based on personal opinions rather than information. The thesis tries to give a research-based evaluation of the techniques for projects with varying demands, this without being irrelevant for practical use.

Also, today's market is challenging in a number of different ways and the users requirements on applications are constantly increasing. The applications needs to be robust, work on different devices, be easy to change and be fast. Because of this it is very important to choose the right tool for the job already during development. There is seldom a "one solution fits all' option available and compromises must be done. Here we saw a need for the decision model, to give insight into what compromises you are making. Even if performance is not the most important factor in a system it is often one of the more important ones, especially since the mobile market just keeps growing. This is why we not only want to deliver the decision model but also provide the separate performance measurements.

To be able to measure the relative performances of the two technologies we implemented a prototype of each API as well as a test environment where the APIs could run and be tested. The creation of the decision model was done simultaneous as the creation of the APIs and the test environment. That made it possible for us not only to base the decision model on literature study but also from the experience gained during the development. The test environment consisted of a web server, an API prototype and performance-logging software running in a container for each of the technologies, load generating software and additional software to log the network load. The load generating software we

implemented was based on Axis' current web application. It fit well to use as a use-case because of its structure, which reminds of many common web applications today, such as Facebook and Netflix. The logging software running in the containers measured each API's CPU- and memory utilization. The other logging software measured the frequency and the size of the traffic communicated between the load generating software and the APIs.

All the measurements showed that GraphQL outperforms REST for our application. Especially how much GraphQL reduced the needed traffic between the client and the server was interesting. GraphQL also showed to perform even better compared than REST when the application grew bigger. But as mentioned earlier performance is only one aspect to evaluate. The decision model shows even more of the differences between the technologies. Put short, using REST results in a much more flexible and adaptable solution while using GraphQL is more restrictive for the developer. GraphQL also needs an implementation that sets some restrictions on which languages that are possible to use. The need of an implementation also affect licensing in ways that REST do not. In pure terms of development using GraphQL requires more work for development and less documentation exists than for REST.

These factors combined imply that GraphQL is a very interesting technology to use for more complex applications that contains a lot of nested data and possibly have the need for efficient mobile clients. A REST solution might be preferable for simple web applications both due to the flexibility but also because of it being more well-known and documented. Using REST also requires less effort to be put into development, which might be important for smaller businesses.

Acknowledgments

We would like to thank our supervisor at LTH Maria Kihl. We would also thank the whole AVHS team at Axis Communications in general and specifically Simon Thörnqvist who were our supervisor at Axis. Simon and the AVHS team gave great guidance and it was a great experience to work with them. Finally, thanks to Axis for letting us do our master thesis project with them.

Table of Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Research question . . . 2
1.3 Purpose, goal and scope . . . 2
1.4 Approach . . . 2

2 Theory **5**
2.1 Software Architecture . . . 5
2.2 Performance . . . 13
2.3 Decision model . . . 16

3 Method **19**
3.1 Defining REST . . . 19
3.2 Performance Measurements . . . 20
3.3 Decision Model . . . 24

4 Results **27**
4.1 Measurements . . . 27
4.2 Decision Model . . . 32

5 Discussion **41**
5.1 Measurements . . . 41
5.2 Decision Model . . . 43

6 Verdicts **51**
6.1 Verdicts . . . 51

References **53**

Chapter 1

Introduction

1.1 Background

This master thesis was done at Axis Communications AB in Lund with the AVHS team. AVHS, *Axis Video Hosting System* is a service that offers alarm monitoring and video surveillance over the internet. Axis offers intelligent security solutions and is a market leading company in the field of network video. They develop cameras, software and applications for network surveillance.
The web and its applications look and work a lot different compared to how they did ten to twenty years ago. Originally and for a long time the web was a place to get web-pages, which can be seen as static documents. A web application used to consist of static pages, which were served from a server. The pages were then retrieved and displayed in the client's web browser. Nowadays web applications are developed in a way where the web is used to "do" things, like watching videos or interacting with other people. This type of web applications often consists of a single page, which continuously is served dynamic content from a web server. The modern application may be seen more as a sophisticated state machine.

Axis wanted to look into the subject of comparing different technologies for implementing web APIs. The reasons for this were mainly:

- *Maintainability* - Axis wanted a system that could keep up with today's fast changing and competitive market.
- *Performance* - The application should be fast and perform well on networks with limited speed.
- *Usability* - It is in Axis' interest that the APIs are easy to understand and consume for third parties.

These three parameters are interesting not just for the AVHS application but for most services communicating over the Internet. It does not matter whether it is a third party who will consume the API or if it is another team within the same company developing a module that will communicate with another module using an API.

1.2 Research question

What strengths and weaknesses are there when working with JSON/Graph APIs and REST APIs and how do they compare to each other in terms of performance, usability and maintainability?

1.3 Purpose, goal and scope

In this thesis we want to investigate two different techniques used for designing web APIs. We want to present a quantitative comparison between them. The Representational State Transfer architecture (REST) has for a long time (together with WS*/SOAP) been the "de facto" standard for implementing web APIs. In recent time new alternatives have risen to fame and a quick search on the Internet about the different alternatives will result in countless discussions about what technique is the best or why one of them is completely useless. The discussions are often subjective, not all, but it is hard to get a clear view of what is actually the difference between the techniques. GraphQL is one of the most used of these techniques and would commonly be the main alternative to a REST API. Because of that we want to compare it to the REST architecture.

The objective of this thesis is not to say which one of the different techniques is better in general, but to perform a quantitative comparison between them and try to find out how the different techniques perform in a common type of application. We want to:

- Conduct performance measurements on API implementations using each of the techniques.
- Present a decision model based on the research made during the thesis and experiences from the development of the implementations. The decision model may be used as support when deciding on what technology to use when designing a web API. It can also emphasize more hard-to-measure differences between the techniques such as maintainability.

The task consisted of three parts:

1. *Literature study*
2. *Develop prototypes and perform measurements*
3. *Evaluate the results and produce the decision model*

The task was estimated to take 20 weeks to finish.

1.4 Approach

A thorough literature study was performed in the beginning of the project. In the study papers about REST, GraphQL, web APIs, Decision models, web server testing and website performance were the main subjects. Papers, blog posts and

discussions around the subjects of web APIs and architectures were also studied. Apart from reading literature, reference implementations of different web APIs on GitHub were also studied. To be able to compare the performance factors like, payload, CPU-usage etc. we developed a prototype application of each the APIs on which we could conduct measurements on. The measurements were performed on the applications when processing a generated load based on characteristics of the AVHS web application.

Chapter 2

Theory

2.1 Software Architecture

Software architecture as a field, discusses the high-level structure of software. It often involves exploring the decisions considering which modules to divide a system into and how the modules will communicate and depend on each other. Here follows some sentences used by Martin Fowler in his book *Patterns of Enterprise Application Architecture*[1] to describe what software architecture is and its characteristics:

- The highest-level breakdown of a system into its parts
- The decisions that are hard to change
- There are multiple architectures in a system
- What is architecturally significant can change over a system's lifetime
- In the end, architecture boils down to whatever the important stuff is

As mentioned there are simultaneously multiple architectures in a system. There are architectures describing the system as a whole and architectures in sub-parts of a system. In the case of web applications the highest abstraction of software architecture could be that it should be a client-server type of application that communicates using the web. Looking deeper into the application, one may look at the server architecture. Which parts should the back-end consist of? Maybe some kind of separate web server as Apache[2] or Nginx[3] could be used. Should there be load balancing? What database should be used? On an even more fine-grained level there are decisions about the actual implementation such as: if programming object oriented, what classes are there? Software architectures have over the years, as software development evolved, become more and more important. The market for software today is extremely fast changing and the demands are getting higher. To be successful in today's market you need to be adapting to its changes, fast! The software architecture is one of the key elements in achieving success. It affect so many parts of the software development: performance, time-to-market, adaptability, maintainability, scalability, robustness etc. Software architecture is not something you "do once" and then it is there. It is

something that requires work throughout the whole lifetime of a software product.
The software architecture is high-level and might be seen as very abstract, but most architectural changes and decisions greatly affects the characteristics of software from a developers point of view. In development, an architectural decision usually affects the maintainability and how easy code is to understand rather than the performance or functionality. Of course there can be an architectural decision affecting performance as well. When it comes to designing web APIs the architecture will greatly affect both soft values like maintainability, understandability and the hard values of performance. So when comparing different techniques for implementing web APIs the softer values, like conceptual differences, needs to be separated from the harder ones like memory usage on the server.

2.1.1 Web API-design

If you have consumed or developed a web API you have probably experienced that often performance, ease-of-use and maintainability are hard to achieve all together. It is often a "you can only pick two" kind of choice, or maybe even only "pick one". If a web API is to be optimized both in terms of performance and ease-of-use, its design should be driven entirely from the needs of the client consuming it.

For example, a client web application has a number of different views that its users can navigate between. On each view there is data being displayed, some explicit to each view and some that are shared. If the design of the API were driven by the clients needs it would expose one endpoint for each view. Instead of having an endpoint for each resource $R_1, R_2...R_n$ there would be an endpoint for each view $V_1, V_2...V_n$ where each view consists of a subset of the resources. This will result in an API that is hard to navigate in and it will not scale well as the number of clients increases.

Spawning multiple endpoints is, even if it is not a good solution, a common one. This solution will not scale well in a larger application where clients have many needs. Apart from making the API hard to navigate and understand, it will also be a challenge to maintain it if the API has to change as the clients change. It is also hard to know what the clients will need when designing the API. These are the main challenges when designing a web API, that it should be both maintainable and easy to use without lowering the performance.

2.1.2 REST

REST, Representational State Transfer, was defined by Roy Thomas Fielding in his doctoral dissertation in 2000[4]. It is an architectural style for implementing web APIs. The derivation of REST is the result of applying constraints on a system, some of the constraints are:

- *Client-Server* - The architecture physically separates the processing capabilities of a system into two different components. A server that offers a set of services and listens for request to those services. The client connects to the

servers and sends requests to it. The server then handles the requests and sends responses to the client[5].

- *Stateless* - The server should not need to keep track of any session state. Each request sent from a client to the server must contain all the information that the server needs in order to understand the request[4].
- *Cacheable* - A response from the server to the client must be able to be labeled as cacheable. This gives the client a possibility to reuse such responses and thus reducing unnecessary requests[4].
- *Uniform Interface* - The different components of the system need to communicate in a uniform way. This constraint introduces a more general interface between the components, which could be for example clients and server. By standardizing communication, the flexibility of using the most appropriate solution is lost. Or as Fielding puts it in his dissertation:

  > Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one that is specific to an application's needs.

- *Layered System* - The system is separated into different layers. One layer only interacts with its closest layers[6]. Think of it as a Unix pipe. E.g. when using `ls | grep txt | awk ' print $1'` , the `ls` and `awk` does not need to know of each other. As long as `ls` and `grep`, and simultaneous `grep` and `awk` (i.e. the closest layers) " communicate " in a correct way, it works.

By applying these constraints REST tries to emphasize scalability of interacting components, a general interface and the possibility to independently deploy components[4]. An API implementing the REST architecture describes the data it wants to expose with resources. In REST a resource is an abstraction of information. A resource contains any type of information e.g. a document, a person or a service. It may be seen as a mapping to a set of information at a given period of time. This may be contrasted to specifying a static document with an URI where the mapping, in this case the URI, is a mapping to information at a single instance. A resource is identified with a URI and the operation to perform on the resource is described with HTTP verbs.

In mathematical terms one may imagine a set S that contains all available data on the server. Figure 2.1 show how different resources contains subsets of S.

For each time instance `t` the resources maps to different subsets of the set S. The resources may be addressed using URIs, see Figure 2.2 that shows an example of how to request a resource. When a request for a resource is received at the server a representation of the subset that is currently mapped to by the resource will be returned in the response.

The REST architecture works very well combined with the HTTP protocol that Fielding also helped specify. REST does, by definition, not depend on HTTP but they are in practice used in pairs. When applied to HTTP the URI of the HTTP

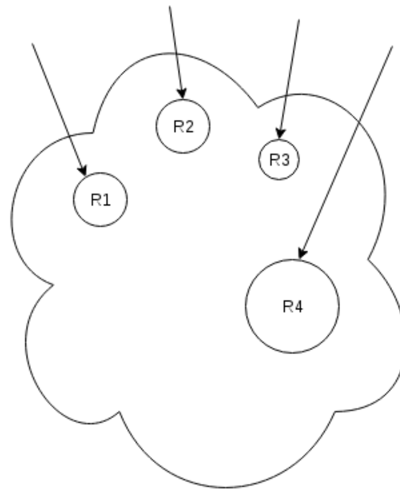

Figure 2.1: Resources as a subset of domain S at a time instance

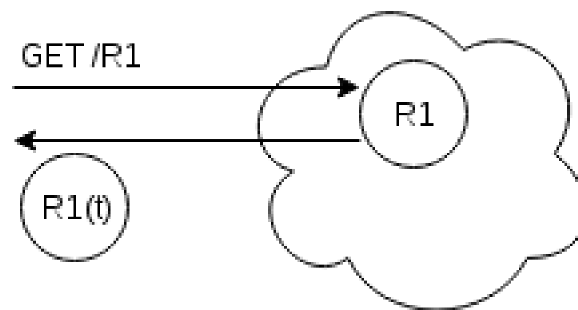

Figure 2.2: Receiving information from a REST resource using HTTP verb and URI

message, see section 2.1.5, is used to specify a resource and the HTTP verb is used to specify the operation on the information mapped to by the resource.

Even though Roy T. Fielding did specify the constraints and more or less standardized REST in his dissertation, it is often used in a "less" strict way. The term RESTful is often used to describe an API that partially conforms to the REST architectural style. Leonard Richardson breaks down the steps of RESTfulness from no-REST to full REST into four levels in his Maturity Model[7].

Maturity Model

- *Level 0* - The API uses HTTP as a transport protocol system for interactions between a client and a server. The server exposes one single endpoint and then the request payload contains the details about the specific resource and operation. This is by all means not RESTful at all.
- *Level 1* - This level introduces resources. By using different URI:s the server may expose different endpoints to the user, each containing a different resource.
- *Level 2* - In Level 0 and Level 1 the HTTP verb is irrelevant for the interaction. This level introduces the HTTP verb as the way to distinguish between different operations. There is some inconsistency among the community about which verbs to use. Lo-Rest uses only two verbs and Hi-Rest uses four verbs.
- *Level 3* - This level introduces a constraint called Hypertext As The Engine of Application State often referred as HATEOAS.

It is not uncommon that an API today only reach level 2 in this model. That is RESTful APIs using the features offered by HTTP on the application level.

Parts of developing a REST-API [8]

A example of the major tasks of designing a REST API:

- - *Resource Identification* - Decide which resources to use to describe the domain.
- - *URI design* - Name the URIs used to access your resources.
- - *Resource Interaction Semantics* - Decide which of the four verbs that should be applicable to each resource.
- - *Resource Relationships* - How are the resources related to each other, reference, ownership containment etc.
- - *Data Representations* - What type of data should be returned. Could be for example JSON or XML.

2.1.3 SOAP/WS*

SOAP, Simple Object Access Protocol, is an XML-based protocol created at Microsoft in 1998. Its purpose is to send and receive information using XML as language to describe the information passed between client and server. SOAP is an extensive protocol and it will not be explained with much detail in this thesis. But knowledge about it is necessary because it is still one of the largest actors among web APIs. The SOAP specification defines a protocol for messaging and addressing different operations between a client and server and as mentioned above it uses XML to define the message architecture. A XML schema is used

so that SOAP engines at both client and server know how to marshal and unmarshal the message content. SOAP does not specify what transport protocol to use like REST does, it is not bound to any specific transport protocol. SOAP focuses on exposing application logic as services instead in comparison to REST that exposes resources/data on which you can perform CRUD (Create, Read, Update and Delete) operations. Due to SOAP's exposing of methods and the use of XML to address these in the messages, a lot of overhead is needed. XML-parsing is an expensive operation in terms of latency and the message creation requires a lot of work.

2.1.4 JSON/Graph-based architectures

The newest trend in the architectural styles of is driven by the likes of Facebook with their GraphQL and Netflix with their Falcor. These technologies take a step back from the REST architecture and would only reach Level 0 in the previously mentioned *Maturity Model* 2.1.2. The technologies remind of older techniques like the previously mentioned SOAP in the sense that they expose only a single endpoint and then make use of a domain specific language to create the request.

The main focus in these architectures is to minimize over-fetching and reducing unnecessary requests. This is done by giving much more power and responsibility to the clients. This thesis will focus on Facebook's GraphQL as the representative for the JSON/Graph-based architectures.

GraphQL was a good choice for the prototype implementation because it had a well maintained implementation in JavaScript that was very accessible. It also represents the features of a JSON/Graph API better than Falcor does. Falcor is a solution that is more suitable for Netflix's specific needs. GraphQL's power comes from a simple idea - instead of defining the structure of responses on the server, the flexibility is given to the client. Each request specifies what fields and relationships it wants to get back, and GraphQL will construct a response tailored for this particular request. The results in that only one round-trip is needed to fetch all the data needed that might otherwise span multiple endpoints, and at the same time only the data that is actually needed is returned [9]. As previously mentioned a resource is to be seen as a representation of data at a given time.

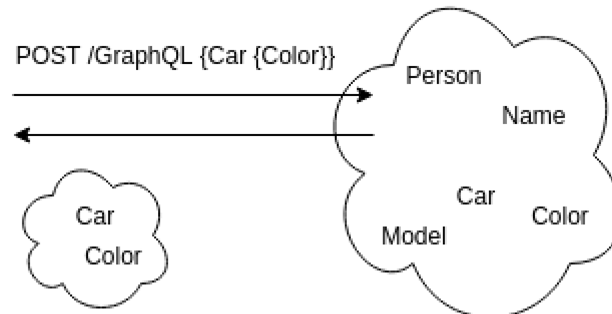

Figure 2.3: Receiving information from a GraphQL server using HTTP and GraphQL query

In contrast to REST, GraphQL is not just an architectural style. GraphQL consists of different parts, a type system, domain-specific language (the query language), execution semantics, static validation and type introspection.

To run GraphQL on a server the GraphQL Core is needed. The implementation of the GraphQL core is language specific. This means that the language used to implement the server must have an implementation of GraphQL to be able to run. Currently there are a number of languages with implementations of GraphQL, with varying maintenance. The main language for the GraphQL core implementation is JavaScript, Facebook maintain this implementation themselves. Other languages currently offering implementations of GraphQL are Ruby, PHP, Python, Java, Scala, C/C++ and Elixir.

The GraphQL Core consists of five components:

- *Frontend (Lexer/Parser)* - Takes a query string and produces an AST (abstract syntax tree).
- *Type System* - The API provided to consumers to describe their own type systems.
- *Introspectrum* - A standardized way of querying the Type System. Often used to ask a GraphQL schema that queries it supports. In the data that is retrievable by introspectrum one can get documentation and information about deprecated fields.
- *Validation* - Validates if a query is valid in an application's schema. For example GraphQL will not allow to pass a string parameter where a enum type is expected.
- *Execution* - Manages query execution. It also handles asynchronous orchestration between queries.

The main idea behind these techniques is that the whole data model is exposed on a single server endpoint. It is then up to the client to "ask", using the

query language, for the parts of the data it wants. The server takes this request, selects the data that is asked for and sends it back as a response. The client will receive only the data it needs and thus lowering the network load compared to a more complex request using REST. An example of a request sent to a GraphQL is described in Figure 2.3.

These techniques were developed because fetching complicated object graphs requires multiple round trips between the client and server. For mobile applications operating in variable network conditions these round-trips are highly undesirable due to their common limitations both in terms of speed and in data usage.

Another thing the creators of the techniques wanted to avoid was over-fetching. Over-fetching is when the client is making a request for some data on the server and the response contains more data than the client needed. For example the user might not want the whole representation of a resource but is forced to ask for it and then parse out the parts that are relevant. This is a problem due to the wasted traffic with unnecessary data sent.

2.1.5 HTTP

Hypertext Transfer Protocol(HTTP) is a stateless application-level protocol and one of the main protocols of the World Wide Web. The current used version is HTTP/1.1 but the new standard for HTTP/2 was released in May 2015 and addresses some of the performance issues coupled with HTTP/1.1. HTTP is the main protocol, due to it being the de facto standard protocol on the web, used to communicate with an API. It is commonly being used to transfer messages between a client and an API and is because of this sometimes referred to as a transport protocol when discussing APIs. When this paper calls HTTP a transfer protocol the meaning of transfer protocol comes from the API context i.e. it is used to transfer the communication between an API and a client. This should not be mixed-up with the more common use of the term that means a protocol that lies in the transport-layer of the OSI-model.

HTTP caching

HTTP includes functionality intended to allow caching of responses. This makes it possible to eliminate the need to send unnecessary requests e.g. requesting an unchanged resource twice. A common way to solve client side caching is for the server to create a tag based on a received HTTP request. The tag gets a lifetime-stamp and is included with the response back to the client. Web browsers may then store the tags with the responses in a cache. Before a HTTP request is sent to a server the web browser calculates a tag in the same way as a server would do. If the tag exists in the cache and if it still is valid the response is fetched from the cache and the request is discarded.

HTTP/1.1

HTTP/1.1 got its first RPC at 1997. Its purpose was to improve the performance of the earlier versions of HTTP. Some of the most important enhancements and features are listed below[10]:

- Persistent connections to allow connection reuse.
- Chunked transfer encoding to allow response streaming.
- Request pipelining to allow parallel request processing. The responses ares still synchronous.
- Improved and much better specified caching mechanisms than HTTP/1.0.

HTTP/2

Google tried to solve some of the problems with HTTP/1.1 with the SPDY protocol[11]. The work on SPDY was then used as a first draft for the HTTP/2 protocol. One of the main purposes for revising HTTP was the problem with how browsers used many TCP connections to send parallel requests to a server. HTTP/2 addresses this problem using unique stream identifiers for each HTTP request stream[12]. By adding multiplexing to the protocol HTTP/2 allows a client to use only one TCP connection to each origin and the server may reply a response as soon as it is processed even if it still have not processed earlier sent requests. The client is then able to distinguish the responses from each other using the stream identifier in the response header.
The key differences between HTTP2 and HTTP/1.x[13]:

- HTTP/2 is binary instead of textual.
- HTTP/2 is multiplexed instead of blocking and in need of ordering.
- HTTP/2 headers are compressed.
- HTTP/2 allows servers to push responses before they are requested into caches at the client.

The fact that HTTP/2 is binary entails less work needed to parse the packets. It is also more compact, which combined with the compressed header gives less overhead compared to a HTTP/1.x packet.

2.2 Performance

From a user perspective one of the most important things when using a website is that it should feel responsive, especially during the initial load phase[14]. The gain from good performance may also be seen from a server perspective. The memory- and CPU-consumption of an application both affect the business in terms of initial cost for physical hardware needed to run the application and also in future costs due to how well the application scale. Scalability is how well/easy an application may be extended with new hardware to handle additional load[15]. An application with good scalability can easily be extended with

a new CPU or more memory etc. without decreasing performance in terms of utilization and responsivity. An application that is not scaling well would use the available hardware in a less efficient way if it is extended with additional hardware.
Developers often try to solve the problem with responsivity by using JavaScript running in the web browser. JavaScript is used to do asynchronous requests to a server, which exposes the data needed by the application. This makes it feasible to load data into the application as the responses containing it returns from the server. Even though this solution to a great extent solves the problem with websites being locked while waiting for the responses, it still limits the users' ability to use the website while waiting for the data from the server responses. The time the client has to wait for the server response, called response time[16], may be seen as the sum of two delays:

1. The time it takes to transfer the request and the response between the client and the server.

2. The time for the server to process the request and return a response.

The server processing time is called latency[16] and it may in turn also be seen as the sum of several latencies, e.g. processing of the request, database look-up and logging. The scope of this report will only cover the first of these, even though database look-ups obviously greatly affects the response time for an HTTP-request. The first delay will from here be referenced to as transfer time.

An API implementation explicitly affects the latency time due to the actual processing of the request. This is due to that a less efficient implementation may take an unnecessary long time to execute certain calculations or processing. This results in a longer latency time than a more efficient solution would give.

The implementation also implicitly affects the transfer time because the transfer time depends on the design. This is due to the design's affect on how many requests the client needs to send to the server. An API is said to have a network profile, that means how it influences the frequency, size and number of different requests. For these reasons it is very important for an API designer to have network traffic in mind when designing an API. By making it possible for the client to get the data it needs with few requests much is won is terms of website performance.

2.2.1 Performance Measurements

To be able to compare a GraphQL API with a REST API in terms of performance the different applications have to be tested. The actual measurements have to be done on implementations of the architectures. The REST architecture acts on a higher abstraction level than an implementation of GraphQL. The performance of the architecture will depend on the implementation, but by choosing the correct measurement points, the effect of the implementation may be reduced. By focusing on relevant measure points the relevant differences of the architectures may be highlighted and pinpointed. Network load in terms of traffic, delays and

connections is affected by architectural decisions. The parameters affected by architectural decisions that will be measured are the amount of requests, size of the data transferred, response time/latency, memory usage and CPU usage. These measurements are relevant during web API design both due to the limitation of possible outgoing connections for a web browser while sending HTTP request and also in terms of cost for the extra server load.

Different hosting providers, e.g. Amazon[17], Digital Ocean[18] often build their business model not only in terms of selling hardware but also by charging the customer based on the use of the network. The use of the network is measured in the total transfer data but also in the amount of requests sent to and from the server.

By implementing the APIs in the same programming language and using the same frameworks the API implementation may be compared in terms of memory and CPU utilization. The memory and CPU utilization affect both the hardware needed for running an instance of the API and also how much traffic each of the instances can handle. This is important both due to pure economic reasons but it also affects the scaling of the API on a given machine. An API that needs a lot of memory and CPU for each connected client either have to be run on a machine with more hardware or be extended with another running instance if the traffic threshold is expected to be exceeded.

The required memory for each API instance can be measured in a deterministic manner. Other running processes on the host machine do not affect the memory usage of the target process. The memory usage for the process can be obtained in two different ways, either with loggers in the server implementation or by using available utilities provided by the operating system. When constructing an environment for measuring the CPU, more parameters have to be taken into account. If many other processes are simultaneously running at the same machine the measured process may get less CPU tics assigned by the scheduler, thus lowering the CPU-usage while instead raising the time needed to process the request.

To be able to control the CPU usage of the process one may run the process in a container or on a virtual machine, as often would be the case today in the cloud computing era[19].

Virtual machine

A virtual machine is an implementation of a computer (or another machine) written in software and intended to run on another machine. The virtual machine runs inside a VMM (Virtual machine monitor), also called hypervisor, which itself in turn runs on a so called host computer. The VMM provides an environment for the virtual machine that mirrors the actual machine and controls the host system resources[20]. This makes it feasible to run both many but also different operating systems on a single machine, each using its own memory and user-space.

Operating-system-level virtualization

Operation-system-level virtualization makes it possible to run multiple different isolated user-spaces on one single computer using containers. Containers are similar to regular virtual machines in many ways and may be seen as more lightweight version of them. This is the solution used to get a deterministic environment for the API implementations that the measurements are performed on.

2.3 Decision model

The term decision model is used in many different fields of work, like, economics, software development, management etc. A decision model tries to take in the relevant parameters in decision making, compare them with each other and help making an informed decision.

When designing a system, there is seldom one perfect solution. There is often a number of alternatives to consider where there are pros and cons with all of them. The decision alternatives are often chosen between in an ad hoc manner. A decision model is not meant to avoid the trade-offs but rather inform of what trade-offs there are and how they may affect your project.
In this thesis, GraphQL and REST are compared to each other. The model used as base for the decision model presented in this paper, is the model presented by Cesare Pautasso, Olaf Zimmermann and Frank Leymann in their paper 'RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision'[8]. They propose a quantitative approach to comparing architectural alternatives. The decision model shows what decisions that have to be made when choosing one alternative over another. Each decision is specified by a number of alternatives. The decisions will be referred to as Architectural Decisions[21][8] and the alternatives to each decision will be referred to as Architectural Alternatives or AAs.

Architectural decisions can be a number of different things, something the designer is bound to when choosing one technique, something that the different techniques handle differently or something where there are a number of different alternatives where the designer has to choose one or more. The alternatives represent the freedom or absence of choice depending on if there are many or few alternatives to a decision. To have absence of choice might sound negative, but it can also be something positive like an existing standard, which is usually not deviated from. J. Tyree and A. Akerman motivate the use of Architectural Decisions in a number of ways[21], one of which is that it *conveys rationale and options*. It is important when reflecting over a decision or motivating it for a client or coworker, to be able to show what decisions were made and what alternatives were considered.

To show what decisions and alternatives that were considered may give a developer guidance on how to proceed with a design. It can also give an API consumer an understanding of how changes in their applications requirements will be affected by the API's architecture. The decisions can also provide a client with assurance that the API with its architecture fulfills their needs.

In *Architecture Decisions: Demystifying Architecture*[21] the question about whether every little detail of a decision should be represented in the model is brought up. The answer is that it should reflect the key elements of a decision making.

The method for eliciting the architectural decisions needed to be made when choosing one of REST and GraphQL over the other is constructed to elicit the key decisions of such a process.

Capturing architectural decisions that have been made in a project is a subject where a lot of work has been done[22]. The work focuses on the retrospective part of decision-taking, i.e. what was done in a project and why was it done. The decision model in this paper, and as emphasized in 'RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision'[8], is to bring forward the decisions that are to come and how they will affect the project.

Chapter 3

Method

3.1 Defining REST

REST is as mentioned earlier a very broad term and RESTful APIs are often called REST APIs. The definition of REST is on an architectural level and can be applied in many different scenarios. To be able to compare the two different architectures a clarification on what the RESTful definition in this thesis actually means has to be made. In Fielding's doctoral dissertation you can find the official definition of what REST is. This is the reason for the term RESTful, most people do not read the whole dissertation. It is a lot of work and skill required to follow every aspect of it. People often do their own interpretation of how to implement REST. They choose to follow some parts REST to best suit their needs. There is nothing wrong in this, but it does lead to some complications. Due to the fact that many implementations that are not REST are still being called REST many confusing and opinionated discussions arise.

This work does not try to validate REST as an architecture but to give decision support when developing a new web API. This decision means that the definition has to have actual relevance with reality.

Most "REST" APIs developed today do not follow strict REST. That means that both the prototypes and the decision model have to take that into account to be relevant.

What will be distinguished on is whether or not REST is followed because of lack of knowledge/skill/time or if it is deviated from because it is the best solution for the problem. The definition tries to include the most important parts of REST and the parts that are most common in implementations like using HTTP as transfer protocol. During this thesis the following statements have been found to be defining what REST is:

- REST uses HTTP as a protocol to transfer requests.
- REST utilizes HTTP verbs to specify operations.
- REST models the domain with resources.
- REST utilizes HTTP URIs to address resources.

Even though the original REST definition does not require HTTP to be used, most implementations use it. In the case that HTTP is not used REST requires a lot more work to be implemented than it requires when using HTTP. Using another protocol for carrying REST messages forces the actual implementation of the communication to be done simultaneous. The developer would need to implement an alternative for the HTTP verb to specify the operation and an alternative to URIs to specify the resource to operate on. HATEOAS (Hypermedia as the engine of state) is not followed in the REST definition from this thesis. This is not because it is unnecessary to use HATEOAS. It is actually a useful concept with many pros if followed. For example, it is possible to change the URI scheme (at least with minor changes) without making the clients that uses the API break. It also gives the clients a possibility to explore the protocol. The reason for HATEOAS not being counted as a requirement for a REST API in this paper is that there is no standard for how to implement HATEOAS and it is also relatively uncommonly used. Even though HATEOAS is not required it will be discussed in section 5.2 how using it might affect the developing process of a REST API.

3.2 Performance Measurements

To be able to measure the relative performance differences between a GraphQL API and a REST API, a prototype for each one of the architectures was implemented. The prototypes were built to mirror Axis' existing API, which is of an ad-hoc RPC (remote procedure call) nature. The term RPC describes a method in distributed computing where the client invokes function calls on the server, which in turn responds with the results of the functions[23]. The decision to mirror Axis' API made it possible to abstract away the database layer and let the APIs only map to the corresponding, already existing, endpoints to fetch data from the database. It has to be noted that the results of implementing this affects the measurements of the response time. Depending on how well the mapping between the existing API and the prototype implementation can be done, the number of requests needed differ.

As an example of this: The prototype implementation models a resource A. This resource conforms to parts of resource B, C and D in the existing API. Then to get resource A, it would take three requests from the prototype to fetch all data needed from Axis' API. Figure 3.1 shows the design of the test system and where the measurements are performed. In the figure it can be seen that the number of requests needed between the Prototype and the Axis API will have an effect on the measurements of response time. It will also effect the measurements of CPU and memory consumption. The effects on CPU and memory will be insignificant because in our tests all of the requests are of the I/O nature and require very little processing.

The measurements of the number of requests, size of requests, CPU utilization and memory consumption can be done in a deterministic way, since we control all parts of the client implementation and the prototype APIs. When measuring the response time however it has to be mentioned that non deterministic elements exist. These elements are the network connection between the prototype

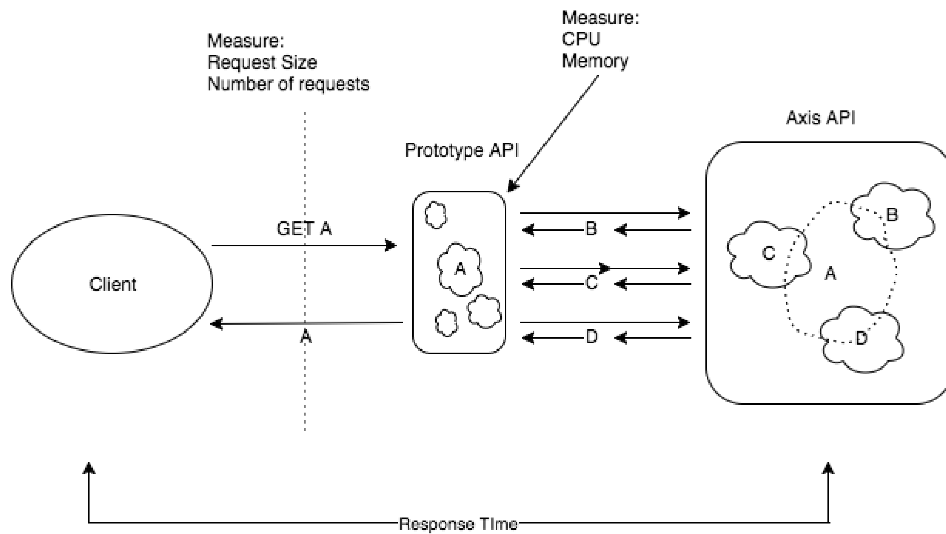

Figure 3.1: System model with the different measurements pointed out

and the existing API, and how the resources are modeled.

3.2.1 Use-case

To get a relevant real world example the use-case was designed based on Axis' web application AVHS. From the AVHS web application one can access the registered surveillance cameras, receive alarms, watch recordings and more. The use-case mirrors what happens in the application when a user have just passed the login page and is loading the index page. On the index page many different resources are needed. The user is presented with a view containing live streams from the cameras that belongs to the user. To be able to do this a lot of information about the cameras and settings are needed. This use-case needs to know of which cameras that are related to a specific site and then retrieve data about them. This data can be available resolution, encoding etc. The use-case is an example of how a real world application could work. In this thesis this scenario will be called "the use-case". To be more specific, the use-case has one surveillance site, which has four cameras connected to it. This set-up was based on how a smaller store, like a clothes store, would use the surveillance application.
This use-case also acts as a requirement specification when developing the prototypes. The prototypes will have to support all the functionality needed to carry out the use-case.

3.2.2 Test environment

The instances of the applications are run in virtual environments during the test. Docker[24] containers are used to create and control the virtual environments. This is to guarantee control over its relative access to the allocated CPU and memory in the container.

3.2.3 Client Implementation

The actual implementation of the use-case mentioned above is written in Python. It consist of scripts that sequentially generates the requests specified by the use-case. There is one script to generate the requests needed from the REST API and one script to generate the requests needed for the GraphQL API. The requests are sent to the API instances running in Docker containers.

3.2.4 Server Implementation

Two different API prototypes were developed to be able to measure how they affect the network performance. The prototypes act as gateways and retrieves the data asked for by the clients from Axis' API proxy. The APIs were written in JavaScript due to access to good frameworks for building APIs and because of Node.js suitability for acting as a web server. In Figure 3.2 below the whole system is illustrated. The GraphQL and the REST application looks almost the same. Where the implementations do differ is inside the Express.js Application in the figure where it says "Query parser / Router". In the REST application, router software is used, which parse the URI from the HTTP request and derives what controller logic to be performed for that URI. In the GraphQL application instead of the Router there is the GraphQL core that parses the query instead of URI and applies the correct controller logic depending on what resource and operation is specified in the query. After the first controller logic layer a number of requests is sent to Axis' API to retrieve the needed data. When Axis' API have responded a response is tailored with the data requested from the client.
One of Axis' computers were used to host the web server and it had the following specification:

- Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
- Memory 16GB (2x 8GB 1600MHz)
- Ubuntu 14.04 64bit

Node.js

Node.js[25] is as mentioned before a run-time for JavaScript. It is based on Google's JavaScript engine named V8, that used in Google Chrome. Node.js uses a non-blocking event-driven I/O-model, making it suitable for I/O-heavy applications with little CPU-heavy work. The application runs a single event thread that registers incoming I/O events to handler functions, which are run asynchronous by a low level thread pool. This mean that a single thread may handle a big amount

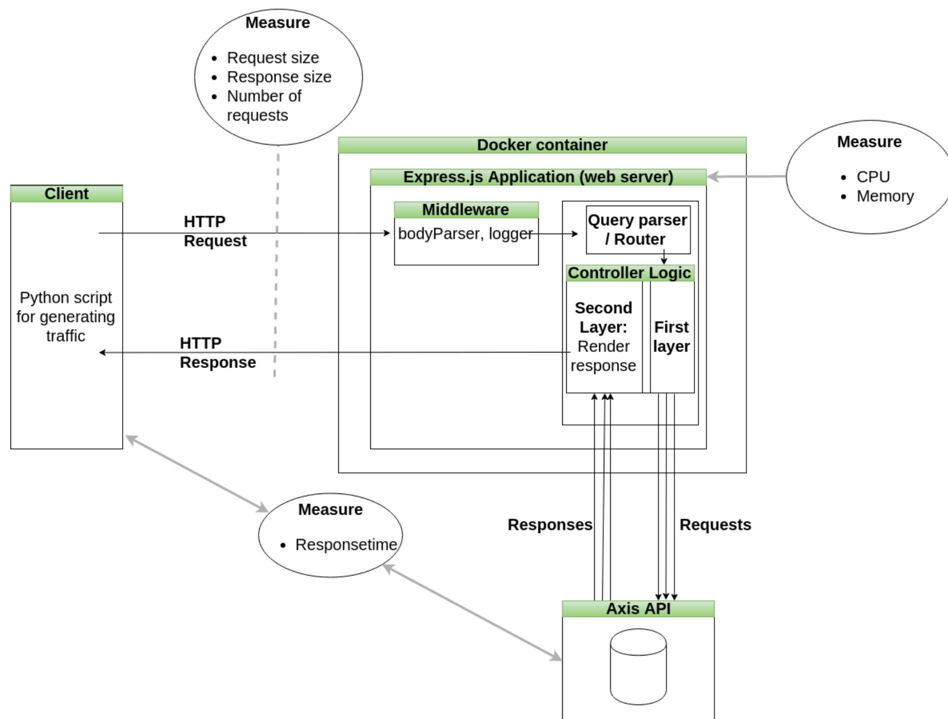

Figure 3.2: System model with a more detailed view of the prototype API

of HTTP connections without blocking due to the short time it takes to register a handle. Compared to other models that starts a new thread to handle each new HTTP connection, which requires a load of overhead, this is very memory efficient. The memory efficiency makes applications running in the Node.js run-time good at handling large amounts of traffic.

REST

The server software consists of an HTTP server written in Node.js that listens to incoming HTTP requests. Depending on the method and URI of the request it routes it to the corresponding controller. The controller sends the request forward to the correct REST end-point of the Axis proxy. Before the requests are routed they are passed through the middle-ware for logging latency, see Figure 3.2. Finally the response from the Axis API is sent back to the requesting client. In the cases where there is more than one request needed to be sent to the Axis API the requests are sent sequentially and when all requests are completed the response for the client is composed from the data retrieved.

GraphQL

This implementation is built with Facebook's implementation of GraphQL using Node.js. A difference compared to the REST implementation is that this web server (also written in Node.js) sends all requests to the GraphQL core instead of routing the incoming HTTP-requests to many resource specific controllers. GraphQL parses the query and addresses the correct resolve functions in the GraphQL Schema. The resolve functions are functions defined for all fields in the GraphQL schema, each returns the data for the specific field. The functions are executed when the corresponding fields are queried and the results are returned in the response.

3.2.5 Technologies and Libraries

During the implementation of the APIs different external libraries and technologies were used. The following section will give a short overview of these techniques and libraries to give a better understanding of how the APIs were developed.

Express

Express[26] is a web framework designed for Node.js. It provides an easy way to implement routing and middle-wares for software serving web applications. It is released under the MIT license.

GraphQL

The GraphQL implementation we used is a reference implementation of Facebook's query language in JavaScript. It provides the ground bricks to build a GraphQL API. This is done by implementing the functionality to building a type schema describing the back-end data and to serve queries to the schema[27].

Docker

Docker[24] is a project that were created to automate the process of packaging applications into software containers. By providing an abstraction for resource isolation features in the Linux kernel, Docker makes it possible to use operating-system-level virtualization.

3.3 Decision Model

As mentioned in section 2.3 the decision model in this thesis is based on the one proposed in " RESTful Web Services vs. "Big" Web Services " by Cesare Pautasso, Olaf Zimmermann and Frank Leymann[8]. The overall structure of the decision model is the same and the same elicitation process was used. The comparison

levels differs some from the ones proposed in their paper to emphasize relevant differences in decisions that needs to be taken when designing a GraphQL API and a REST API.

3.3.1 Elicitation of Architectural Decisions and Alternatives

The goal with the elicitation is to find the relevant parameters to compare the different ways of implementing web APIs from. Relevant parameters refer to the key decisions needed to be made when choosing one technology over the other, not every detail that separates them.

The elicitation process is conducted in the following way:

1. Screen reference information and compare what problems the different techniques address and how they are "solved".
2. Develop different sample scenarios of the decision making. Record the number of architectural decisions made and development steps required when using GraphQL and REST.
3. Conduct interviews with people who can relate to either developing a web API or consuming one. Elicit from them what they think is important when creating/consuming web APIs.
4. Create one decision model for each of the two integration types based on the results from step 1 and 2.
5. Compare the two models from step 3 to see that they address the same design issues. This should result in one decision model.
6. Measure and compare number of decisions and number of options per decision.

3.3.2 Comparison Levels

To categorize the decisions so that the model will be easier to understand the decisions elicited are divided into four comparison levels. The first three of the levels are proposed in the " RESTful Web Services vs. "Big" Web Services "[8] and they were found relevant and appropriate to use also for this comparison. The last one was based on the gained experience from the implementations of the APIs for the measurements.

1. *Comparison of architectural principles*. What principles define each technology and what requirements do they set on the implementation. When describing architectural principles buzzwords like *REST have a strong separation between client and server* are often used. A comparison on this level is often subjective and not enough to emphasize the relevant differences between the technologies[8] therefore more comparisons are needed.

2. *Comparison of conceptual models*. What conceptual decisions are required when following the different techniques. An example of this would be that in REST one identifies resources while in some other architectural style one might

identify operations instead. This shows the conceptual differences when designing an API of each of the techniques.

3. *Comparison of technology decisions*. Compare how the two techniques can be technically realized. This could be for example what transport protocols one can use or in what format one may send the payload. This level show which different technology decisions that have to be taken while designing APIs of the techniques.

4. *Comparison of implementation technology*. Shows the decisions that have to be taken about what technology to use during implementation of APIs of the techniques. This could be which licenses third party libraries that have to be used are released under.

Chapter 4

Results

4.1 Measurements

The measurements were carried out on a running Docker image containing the server implementation. Software developed for logging CPU, memory and request characteristics were running on the images. To simulate traffic to the server the traffic generating python scripts were running in parallel. As may be seen in the plots, the GraphQL APIs required less time to process the traffic. Since the scale on the x-axis is in seconds and the GraphQL API required less time to process the traffic, its line ends earlier than the REST APIs. This is because of that the test is performed in the manner that there is a predefined amount of requests to be served as fast as possible instead of constantly serving requests.

4.1.1 Memory

The following plots show the memory used by each of the API applications. The number of instances generating traffic sent to the server was increased with time during the measurements. At t=0 there was one instance generating traffic and then the number was increased step wise up to ten instances generating traffic in parallel. Figure 4.1 shows the used heap memory of the API processes while handling the load where the upper graph is for the REST server and the bottom one for the GraphQL server. As expected and presented below the GraphQL server utilizes more memory than the REST server. This is due to the need of the GraphQL core on top of the functionality needed by the REST API.

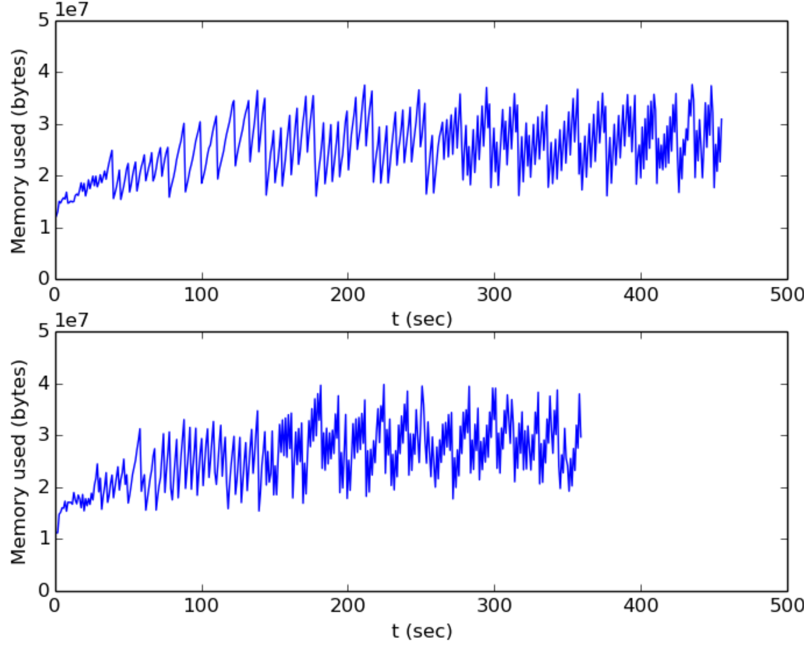

Figure 4.1: Memory measurements. Top graph is for REST and the bottom one is for GraphQL

4.1.2 CPU

The CPU measurements were performed in another simulation than the memory measurements. This was done to reduce the logging process' effect on the measurements of CPU utilization and vice versa. Figure 4.2 shows the CPU utilization for each of the processes. For better visualization of the data a sixth degree polynomial were fitted to each curve using non-linear least squares with no weighting. The points on the new curve are calculated by the following formulas:

$$S = \sum_{i=1}^{m} r_i^2 \tag{4.1}$$

where S is the sum of squares of the residuals (r_i), that is the sum of all errors that in turn should be minimized, the residuals are given by:

$$r_i = y_i - f(x_i, \bar{\beta}) \tag{4.2}$$

y_i are the values from the original measurements and β are the coefficients of the fitting function:

$$f(x, \bar{\beta}) = \beta_1 x^6 + \beta_2 x^5 + \beta_3 x^4 + \beta_4 x^3 + \beta_5 x^2 + \beta_6 x + \beta_7 \tag{4.3}$$

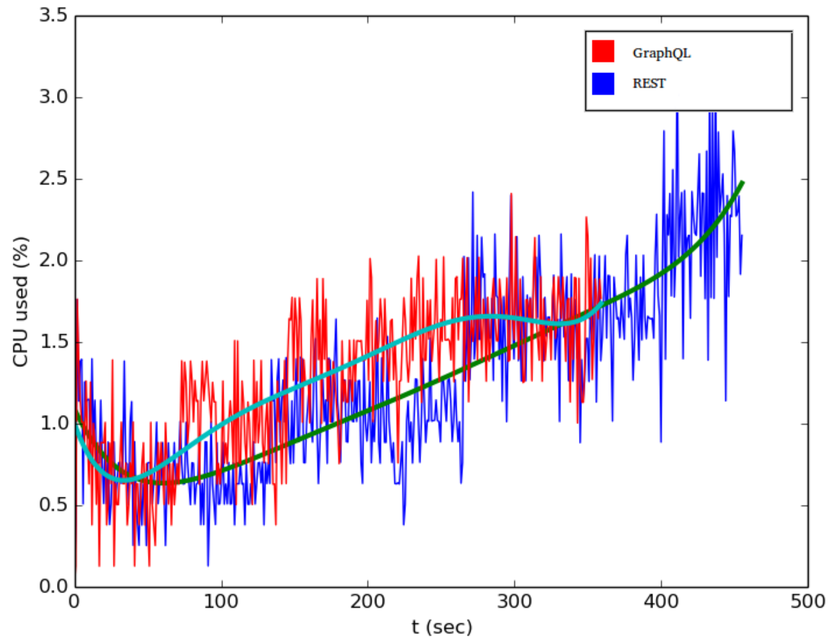

Figure 4.2: CPU measurements

A number of different polynomials were tried out as fitting functions but the sixth degree above gave the best result.

4.1.3 Request characteristics

Wireshark was used to log all the requests and responses together with their content. The amount of requests needed to be sent from the client to the server to complete the use-case is shown in Figure 4.3 below. In the graph it can be seen that the number of requests needed to fetch all data grows faster with the REST server then with the GraphQL server when the number of resources is increased.

It is mainly in the data received that the two implementations differ a lot. This is because in GraphQL you ask specifically for the data you want instead of as in REST you ask for all the resources containing the data. This was expected since one of GraphQLs main features is that you have client specific queries and therefore no unnecessary data is sent. Also less request and response being sent leads to less overhead data being transferred. The measurements of the size of the received and sent data for 100 clients performing the use case once (with four cameras) are presented in Figure 4.4.

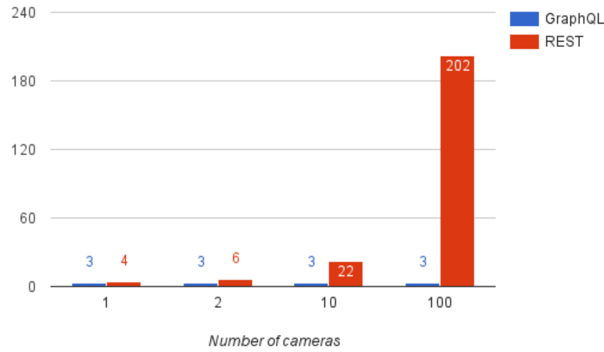

Figure 4.3: Number of requests for varying number of cameras on the site.

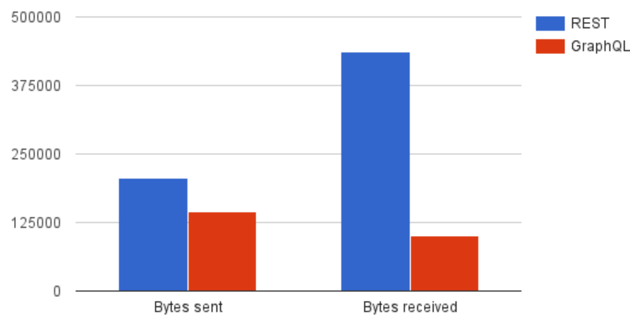

Figure 4.4: Total number of bytes sent and received for 100 clients performing the use case one time each.

4.1.4 Response time

This is a result of adding up the response times for the use-cases. This measurement were also used to calculate the average response times for REST and GraphQL. It is worth mentioning that the requests are executed in sequence i.e. the previous requests must all have been processed before the next iteration starts. In Figure 4.5 one can see how the performance scale better with the GraphQL solution when increasing the number of iteration, this due to less requests needs to be sent.

The average response time was calculated by running the use-case 1000 iter-

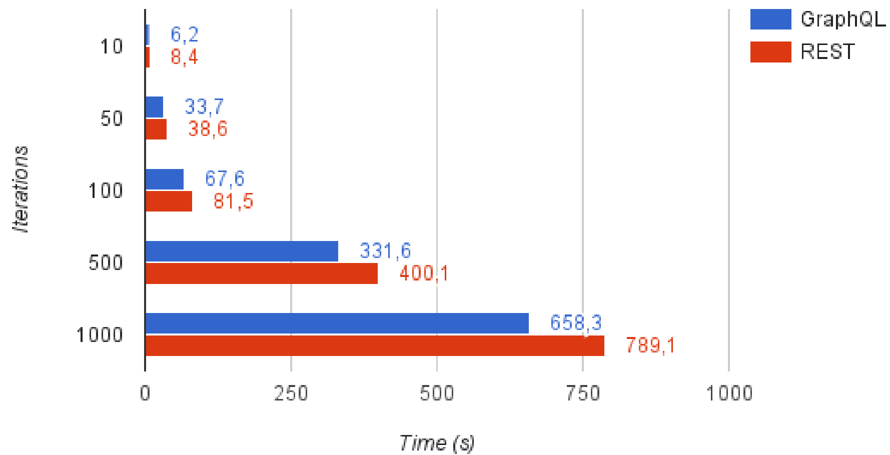

Figure 4.5: Response time for 10-1000 iterations of use-case

ations and calculating the arithmetic mean of the measurements:

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_{1000}}{1000}$$

where $[x_1 \cdots x_{1000}]$ are the measurements of the response time for each iteration. The mean response times for the two cases are approximately 789.1 ms for REST and 658.2 ms for GraphQL. This results in a difference of 130.9 ms in response time between the GraphQL and the REST API.

4.2 Decision Model

The following results are the decisions needed to be taken when designing an API for each of the technologies. The different alternatives and other relevant notes for each of the technologies are also mentioned. The following sections are derived from the literature study but also from experience of working with APIs during the implementation of the prototype APIs and from interviews with architects and developers at Axis. In each subsection one part of the decision model is presented and then follows a more detailed description of each of the fields in the model. Each table of the model consist of the decisions and below them the different alternatives. The numbers of alternatives are summarized for each decision and technology and included in the tables.

4.2.1 Comparison of architectural principles

The overall structure of an architectural style is defined through its defining principles. The defining principle can be seen as the broad picture of the architectural style and how the style solves its requirements and goals. During the investigation the two following principles of the design were found relevant and comparable in an objective way. In the table below, *Comparison of architectural principles*4.1, the findings are presented.

Table 4.1: Comparison of architectural principles

Archtectural Principle and Aspects	REST	GraphQL
Protocol Layering	1	**1**
Http as transport-level protocol		x
Http as application-level protocol	x	
Loose Coupling (aspects covered)	2	3
Uniform interface	x	x
Stateless	x	x
Client Specific Querys		x

Protocol Layering

Protocol Layering refers to whether the HTTP protocol is used as a transport or application protocol. This terminology is not based on the OSI-model where HTTP per definition is an application-layer protocol. Here the term application protocol mean that the protocol not only is used to transfer data to a "higher" layer such as the GraphQL parser. The previously mentioned example uses the protocol as a, here called, transfer protocol. HTTP being used as an application protocol is a de facto standard for RESTful APIs so in this paper it is assumed that HTTP always is used as an application protocol for REST APIs. When using GraphQL one are not bound to use HTTP as transport protocol. Although most people implement it with HTTP because most frameworks and tutorials does.

Loose Coupling

Loose coupling can be defined in different ways and the meaning depends on the context. In terms of web APIs when using a client server solution, which this thesis is about, loose coupling focuses on the interaction between the provider and consumer of a service. When communicating between client and server there is almost always a protocol for how the communication is carried out. Since client and server depend on each other in that manner the term is loose coupling, not "No Coupling". In this context loose coupling would mean that changes made in the client or server not will affect the other party while changes to the actual protocol will have to be communicated somehow.

To achieve loose coupling it is required that both consumer and producer understand and respect a common protocol and semantics. Both techniques emphasize the importance of this point and a distributed system that is loosely coupled can be achieved with both. REST is defined by a number of principles and a stateless, uniform interface and client-server separation ensure a loose coupling between client and server. A uniform interface is achieved by applying four interface constraints[4]: *identification of resources*, *manipulation of resources through representations*, *self-descriptive messages*, and *hypermedia as the engine of application state*. Fielding lifts the fact that there is a trade-off being made to having a uniform interface instead of an interface that is specific to an applications needs. To transfer information in a standardized form instead of in a form that is specific to an applications is often not optimal but that is the price one have to pay for the uniform interface. The trade-offs being made when designing web APIs boil down to just this problem and that is one reason to why strict REST might be deviated from. The designers of GraphQL do not name their design principles in the same way but they support the same principles. A uniform interface is achieved by having a uniform language, type system and client specific queries. With client specific queries GraphQL avoids the trade off by not being able to transfer information in a form specific to the application.

4.2.2 Comparison of conceptual models

This section of the decision model 4.2) show the conceptual decisions that must be made when deciding to use one of the architectural styles. The model also show which alternatives that may be chosen when making the decision.

Table 4.2: Comparison of conceptual models

Architectural Decisions and AAs	REST	GraphQL
Integration Style	1	1
Shared Database		
File Transfer		
Remote procedure call	x	x
Messaging		
Distributed object communication		
Resource Identification (maturity level 1)	1	1
Do-it-Yourself	x	x
Design of resource addressing	2	1
Nice URI scheme	x	
No URI scheme	x	
Schema (JSONgraph)		x
Resource Interaction Semantics	3	1
Maturity Level 0 (POST only)	x	
Maturity Level 2 (Lo-Rest GET/POST)	x	
Maturity Level 2 (Hi-Rest all four verbs)	x	
Query language		x
Resource Discovery	1	1
HATOES (Maturity Level 3)	x	
Introspection		x
Documentation	1	1
Generated Schema		x
Tools (like swagger)	x	
Do-it-yourself	x	

Integration Style

Integration styles are different methods to make different parts of an application work together. Both REST and GraphQL can be used for RPC style communication. Other integration styles like the ones presented in the article *Introduction to Integration Styles*[28] by G. Hohpe and B. Woolf, and the examples listed in the decision model, are not where they are meant to be used. The different integration styles each have their strengths and it is important to point out that each should be used where their strengths can be utilized.

Resource Identification

This decision involves how to describe the data at the server in terms of resources. The resources could be products, customers, orders, baskets etc. depending on your domain. The process to identify those is an ad-hoc process requiring domain knowledge and experience of designing web APIs. Imagine implementing an API for a bank application, it is quite easy to realize that customers and accounts should be two of the resources, but whether a transaction between two accounts should be represented as a resource is less intuitive. It requires both domain and technical knowledge to make those decisions. GraphQL is facing the same challenges as REST when designing resources, relations and operations.

Design of Resource addressing

This describes the different possible ways to design how one should address different resources. One of the principles in REST is addressable resources just like the different websites are addressed on the web. To achieve this REST uses URIs, and if following good REST practice, they have a "nice design". Designing a nice URI could be for example that the URI should be short, readable, predictable, permanent and tied to a resource. The format of an URI is well defined, the design is not. REST practice defines guidelines for the so called "nice" URI design so that it is possible to "guess" what a URI for a specific operation on a resource will look like. In GraphQL instead of URIs there is a query language.

Resource Interaction Semantics

In object oriented programming, interfaces are often used to decouple classes from their implementations. A client-server solution that conforms to the REST architecture should work in the same way. One part of having a uniform interface is to have a standard for how resource manipulation is carried out and REST leverages the HTTP verbs as standard for that. It is similar to how an interface in Java would work. As an example, imagine an interface called Shape, which has a method called getArea(). The programmer would then know if the interface is implemented, the method getArea() should be the method returning the area for my implementation of Shape. So in our setting, REST architectural style is the interface. In GraphQL the process is more ad-hoc. Mutations and operations are designed by the developer so there is not a fixed set of names for them as in REST where HTTP verbs are used. This makes the introspection feature of GraphQL very important. The query-language is provided by a context-free grammar but the fields themselves are defined by the type system.

Resource Discovery

Instead of having a documentation that tells what resources and operations there are, resource discovery enables the feature of being able to "ask around". From a given starting point to the API it is possible to ask for what relations between resources and possible operations. In REST, resource discovery is solved with HATEOS. In GraphQL there is something called Introspection, the consumer can query the API itself and its types with the same query language used to get or mutate data. Resource discovery might be of more importance in machine-to-machine communications where there are no humans involved and a static interface will make the API fragile.

Documentation

REST provides no support for documentation. All documentation need to be created manually or by using external tools. With GraphQL there is the feature for introspection. The client can access the information about the API by querying the schema itself with the same query-language used for accessing and modifying data.

4.2.3 Comparison of technology decisions

Table 4.3 shows the comparison of how the two technologies can be technically realized. It shows which decisions a developer needs to take from a technical standpoint and which the alternatives are for the decisions.

Table 4.3: Comparison of technology decisions

Architectural Decisions and AAs	REST	GraphQL
Transport Protocol	1	4+
HTTP	x	x
Websockets		x
TCP		x
UDP		x
Payload format	3+	2
JSON	x	
GraphQL (extended JSON)		x
XML	x	
Response-data determined by server	x	
Response-data determined by client		x
Resource/Service Identification	1	1
URI	x	
Query		x
Transactions	1	1
Do-it-yourself/Not available	x	x
Versioning	3	2 (if needed)
Explicit route	x	x
Accept Header	x	
Custom Header	x	x
Caching	2	1
Leverage HTTP cache	x	
Do-it-yourself	x	x

Transport Protocol

This is the protocols one may use to transfer messages between a client and a server. The protocols does not need to be protocols in the transport-layer of the OSI-model, it is as mentioned before most common to use HTTP. GraphQL serves queries on one single endpoint independent from what transport protocol is used to deliver the query. As declared earlier in this thesis it is assumed that REST uses HTTP as transport protocol and therefore HTTP is the only option given here.

Payload Format

This decision is about which format should be used for the payload of the messages. REST is flexible and may return data of any structure. GraphQL needs to follow GraphQL's JSON-like structure. In REST the API designers implement the way the server response should be formatted. This could be solved by adding the support to query fields with parameters but it violates the REST architectural constraints. In GraphQL the client queries the data and "designs" the structure of the response.

Transactions

There is no support for transactions in neither REST nor GraphQL. When transactions are needed the functionality has to be implemented by the developer.

Versioning

There are two common ways of solving the problem with versioning when using REST. Either by explicitly stating the desired version of the API in the URI, i.e. the API designer has to add the version to the URIs. The other way is by using Content negotiation, it is performed by providing the desired version in the Accept field in the HTTP-header. The field is then controlled in the server and the correct version of the service is executed. You can also imagine that there can be a custom header to specify the API version but since the Accept header field is supposed to be used for this kind of tasks that option is left out.

Due to the fact that clients ask explicitly for the resources and fields they want to retrieve there is no need for versioning the API unless there are changes that will break the existing resources.

Caching

By caching messages, both on the client-side and on the server-side, one may save many unnecessary requests. Support for caching is therefore a good way to save both network load and CPU power for the application provider. Both REST and GraphQL APIs are stateless. Caching is therefore managed at the client. REST provides server-side caching since it uses the HTTP protocol, which makes it feasible to leverage its caching by using ETags. GraphQL does not offer any support for caching.

4.2.4 Comparison of implementation technology

Table 4.4 shows the decisions about which implementations and tools to use when implementing APIs built on the styles.

Table 4.4: Comparison of implementation technology

Architectural Decisions and AAs	REST	GraphQL
Languages with maintained implementations	n/a	5
Javascript		x
Ruby		x
PHP		x
Python		x
Java		x
C++		
Client-side frameworks	1	2
Relay		x
Do-it-yourself	x	x
Licenses	n/a	2
MIT		x
BSD		x

Languages with available maintained implementation

GraphQL is quite a new project but there are already maintained implementations of GraphQL for many programming languages. Maintained is a vague term, but to be considered maintained in this thesis the project needs to be regularly committed to and worked on by at least five developers. As mentioned earlier REST is an architecture so there is no such thing as an implementation of REST in the same sense as a GraphQL implementation. As long as the language has support for handling HTTP messages, the language may be used.

Client-side frameworks

When developing applications one might use some kind of client-side framework for handling data sharing between the client and the server. The framework is used to give the illusion that the client side code interacts with a data structure stored on the computer. All interaction with the server, as creating requests for fetching data is then handled by the framework. For GraphQL APIs, Facebook's framework Relay is often used for handling the interaction. It is not as common to use a client-side framework for handling of REST request, but larger "full-stack frameworks" often tries to capture the data sharing.

Licensing

REST is as mentioned before an architecture. Because of this the designer does not need to take any licenses into consideration. The grammar of the GraphQL languages is "free" but the reference implementation of GraphQL is released under the BSD license. Implementations in other languages are released under a

varying set of licenses. Most of the implementations found are under BSD and MIT licenses. The possibility to go with a community based implementation depends on the case, but due to the actual grammar being free the possibility for the developer to implement a " do-it-yourself " solution always exist. The previously mentioned Relay framework for data sharing is also released under the BSD license.

Chapter 5

Discussion

5.1 Measurements

The measurements presented is discussed more thoroughly in this chapter. They are also related to the real world to analyze how they relate to the perceived and actual performance of a web application.

5.1.1 Memory

Implementing the server in Node.js means that the server is single-threaded due to the nature of the language. This is very suitable for some applications but not for others. Typically Node.js is very effective for applications with a lot of I/O-processing, such as an API for a CRUD application, while more calculation heavy applications might suffer in performance. We expected to see larger differences in memory consumption between the implementations. The reason for the difference being fairly small is first of all because of the need for more generated traffic. The second reason for the small difference were that in Node.js, as explained above, only one thread handles all requests. If we would have implemented a web server using one thread per request, which is not uncommon, the difference between REST and GraphQL would probably have been larger. A Node.js process by itself needs a lot of memory. Therefore the memory cost of keeping up the amount of connections that we had did not emphasize that much of a difference between the APIs in memory consumption. If a web server solution where each new connection would be run in a new thread were used the benefits of using GraphQL would be more in terms of memory consumption.
The sewing machine characteristic of the graphs is a result due to garbage collection in the V8 engine. The garbage collector does both small and big "clean-ups", which gives many small memory releases between every big garbage collector iteration. We tried running instances of the APIs using both different parameters and available run-time functionality of Node.js but were not able to control the garbage collector and get better plots. We also realized that using the garbage collector in a non-conventional way would only give unusable data, after all nobody would ever implement a web server without using garbage collection and the data would thus be irrelevant.

5.1.2 CPU

Our measurements of the CPU utilization were also a bit disappointing. We expected to see more of an overhead cost when using GraphQL compared to REST. We also expected GraphQL to be faster during higher load due to it having to process less requests. The pattern may be seen from the data in Figure 4.2. With more traffic generated it would probably show more difference than we got in our test. We did not have enough hardware to set up a test environment to be able to generate that much load to the API instances and therefore the low CPU consumption in our measurements. It has to be noted that applications with low needs of processing power, which instead spends much time on I/O, often can handle a lot of traffic without pressuring the CPU much. It was a bit optimistic of us to expect that we would be able to generate enough traffic without access to external load generating services, to pressure the CPU.

5.1.3 Page Speed

The measurements on response-time in our experiments resulted in a difference of 130.9ms, where GraphQL was the faster implementation. In the use-case where response times were measured, four cameras were used. This resulted in ten requests needed to load the index page using the REST API and one needed when using GraphQL. When Netflix explained why they created Falor, they said that using RESTful approach for their API resulted in having almost 100 requests needed to load the index page. This was a result from that the Netflix application required so many nested resources when working with REST[29]. The more nested data the application need to load, the greater the gain in performance will be by going with a GraphQL API instead of a REST API. This behavior is seen in Figure 4.3, it show that the number of requests needed when increasing the number of cameras on the site. In the REST implementation the requests grows linearly whereas the requests needed in the GraphQL implementation are constant.

130.9ms might sound like a small difference but research shows that a negative effect on user experience can be seen by introducing delays of that magnitude. In a study conducted by Google it was stated that when exposing users to a 100ms to 400ms delay when loading the search result page, their number of searches went down by -0.2% to -0.6%[30]. This impact was from slowing down the page for a very short period of time. If the users were kept being exposed to slower loading times it could be seen that the users were doing fewer and fewer searches the longer they were exposed.

Depending on the application there is more or less to be gained in terms of page speed. In applications that need to communicate a lot of data rather than performing processing, calculations etc. much can be gained from using a technique such as GraphQL or Falcor instead of REST. This is because most of the performance issues are inherited by I/O rather than processing time.

5.2 Decision Model

In this section we summarize our own experiences from working with the techniques and discuss the different parts of the decision model. Together with this we discuss how the differences will express themselves in different applications.

5.2.1 Comparison of architectural principles

Protocol Layering

When implementing REST you are not bound to HTTP but almost every RESTful application use it to transfer the requests. When developing the prototype APIs, the approach of not using HTTP as transport protocol were tried. This resulted in many lines of code for all the extra parsing and logic needed. This approach was abandoned due to all the extra work needed while losing the caching provided by HTTP. This experience led to this thesis' definition of REST where using HTTP for transferring the requests is a requirement. In some applications like a chat server it might be more suitable to use web sockets instead of HTTP and therefore the question of "Can I use REST with web sockets?" arise. The verdict here is that going with web sockets is probably a good idea but to try and force a REST solution on top of that is not recommended.

Loose Coupling

Loose coupling is quite an abstract term. It affects how an API consumer will be impacted by changes to the API. If a change to a resource is made in the API, the question is how the client using it will be affected. If it is a REST API it depends on what the changes are and if HATEOAS is being used. If HATEOAS is being used, a change to a resource URI will not affect the consumer unless the change affects the entry point to the API. Changing the resource itself by either adding or removing data will affect the consumer and potentially break their application. Using GraphQL it is safe to change data as long as data not is removed without affecting the client. This is also the way Facebook recommend working with GraphQL APIs, "never remove content from the schema only add". All techniques for implementing APIs struggle with this problem and because of the consumer/producer character of web APIs it is impossible to avoid it. GraphQL handles this problem in the most secure way according to our experience. The schema might be a bit bloated in the long run though and require good practice from developers to keep it structured.

5.2.2 Comparison of conceptual models

Integration Style

There are many different ways of building communicating applications, some but not all of them are listed in the model. REST and GraphQL are both meant to be used for web services and are therefore evaluated for that and no other

integration styles. It should not be forgotten though, that there are many different kinds of applications with different needs and constraints. The best integration style depends on the domains requirements[31].

Resource Identification

Resource identification is one of the parts in designing a web API that is considered a handcraft and the technologies does not offer any help here. REST APIs often require a lot more effort than designing a GraphQL schema, at least if efficiency is highly prioritized. The design and choice of resources greatly affects both performance and the usability of the API. Due to GraphQL's query based solution for accessing information from the server, it will not affect performance how the resources are defined. But the definition of the schema will still affect the usability of the API. If the fields do not conform to what the user expects the usability will be affected negatively.

Design of resource addressing

The REST maturity model is mentioned and often referred to when discussing REST. Depending of what maturity level the API is at, the resource addressing differs. REST maturity level 0 has no URI scheme and level 1-3 use what we have referred to as a nice URI scheme to address resources. GraphQL uses a schema and query language to address resources and it does not work the same way. No decisions have to be made about which style to use when designing resources in the GraphQL schema. The problems faced when designing a GraphQL schema are instead mainly about naming resources. This has not as big impact on the overall API as e.g. deciding which maturity level to use when designing a REST API.

The challenge with REST is to construct nice URIs and it requires some work to come up with a design that is good. When creating our prototypes we spent a lot more time creating the GraphQL schema than the equivalent REST endpoints. This was because the challenge in REST is more of finding a good practice for how to construct the URIs. Once a standard way of designing the URIs is set the work is very simple. In GraphQL actually implementing the schema requires a lot of work and thought, especially for inexperienced (with GraphQL) developers. But when addressing a resource in GraphQL the query language is convenient to work with because it is a well-defined syntax.

Resource Interaction Semantics

Using REST one is limited to use the HTTP verbs. The number of existing HTTP verbs might be limiting for some applications. As an example: A bank application has a resource that is accounts. How would a transaction between two accounts be carried out? It is not possible to post to one account at a time since that would require a state in the server. In most REST practice, the recommendation is to have a resource representing a transaction which one can post to. From our experience the verbs are usually not limiting and when they are, designing resources

to represent the desired functionality will solve most problems. In GraphQL the developer of the API decides which operations that are available and what they are called. This results in more freedom when creating the API but there is no standardized naming convention either.

Resource Discovery

A pure REST API utilizes HATEOAS, which introduces a light version of resource discovery. The server responses not only include data or response status but also links to all related resources. Because of this the client only needs to know of one fixed entry URL to explore the API. What is missing here is that in HATEOAS one are not told what methods the related resources are open for. The rest of the possible interactions will be given by the server according to Fielding[4]. An example of a server implementing HATEOAS responding to a request asking for a person with a brother relation, the response would look as the following:

```
{
    "name": "Erik"
    "dateofbirth": 19890810
    "links": [ {
        "rel": "brother"
        "href": "http://localhost:8000/person/2"
    } ]
}
```

GraphQL per design includes a way to introspect the data and operations available on the API. This makes it possible to discover all resources on the server and the operations available on them. The main difference is the availability. The REST/HATEOAS way of solving resource discovery better fits a machine. If third party developers should use the REST API, extra documentation would be preferred. GraphQL in contrast provides an overview of the server schema to the developer without the need to follow hyperlinks, as in the HATEOAS case.

Documentation

The subject of documentation is where REST and GraphQL are hard to compare since REST is an architectural style and GraphQL more of a framework. What REST does offer is HATEOAS, which can be seen as a light documentation. From our experience and judging from discussions and tools like Swagger, many others feel the same way. To get a documentation over REST API one has to create it manually or use a tool such as Swagger to generate it. This can be tedious work if done manually and if tools are not used it requires a lot of extra work. Where the effort really increase is when maintaining a system, then every change will have to be verified in the documentation and updated if needed. When using GraphQL one has the possibility to get an overview of the schema using regular queries. This works well as documentation over the available resources and available operations on the server.

5.2.3 Comparison of technology decisions

Transport Protocol

GraphQL mostly uses HTTP as well but remains independent from what protocol is used for transport. REST is not bound in theory to HTTP, but the de facto standard way is to use it. Thus as a ground for decision making REST has to be seen as bound to HTTP. To use another transport protocol the developers would have to specify semantics both to communicate which operation to use and which resource to address on their own. It introduces a lot of work, both for initial implementation and for maintenance. It is not seen as a fair alternative to use anything else than HTTP in REST. As GraphQL always use the payload of the HTTP-request to transfer its query it is not as much work to do for the developers when changing transport protocol.

Payload Format

In most of the subjects until this one, REST has been the more restrictive one. Regarding what format is used for the payload, REST is more liberal than GraphQL where one is bound to their JSON-like format.

Transactions

Neither REST nor GraphQL offer any built in support for transactions. The REST solution would be to create a resource that represents a transaction. This is not possible in GraphQL in a natural way. When the GraphQL parser processes a query it iterates through each field of the query and tries to match it to a function, which is defined in the GraphQL schema. The search for the function in the GraphQL schema is done using breadth-first search. The result of this function is then returned in the response. Due to the nature of GraphQL all of these functions, also called resolve-functions, will be executed sequentially. If one of the latter fields fails, the previous resolve-function will already have been executed and thus will result in non-ACID (Atomicity, Consistency, Isolation, Durability) behavior. There is no good solution that fits all scenarios here. It is possible though, to define the schema such that all mutations of the data will be done in a transaction. This solution could possibly lead to unnecessary locking of the database and thus lead to performance issues. Another solution would be to use another software, which could handle the transaction part of the mutation. It would increase both complexity and maintenance effort of the software to go with any of the proposed solutions. Due to this fact we believe that GraphQL is not suited for an application that requires guaranteed ACID transactions. GraphQL's strength lies more in the CRUD domain.

Versioning

To version an API means how changes to the API are handled and communicated to the consumers. This is in many applications very challenging since domains and requirements change rapidly. This is one of the more "noisy" discussions

about REST, kludged with opinionated conclusions. We think this is the result due to the absence of a commonly adopted best practice for how to do this. When working with REST there are three ways of versioning the API[32]:

- Encode the version into the URI
- Have a custom header for API version like: ”api-version: v2”
- Use HTTP Accept Header: Accept: application/vnd.api.v2+json

All of these options have their disadvantages but as Troy Hunt writes: *It is about having a stable contract*. The first one, to encode the version into the URL is the worst one from a REST perspective. When encoding the version into the URI, it no longer represents the resource. Having the version in the URI is the easiest to work with from our experience even though it is the least correct way in terms of REST.

GraphQL tries to solve the problem of versioning by encouraging to have an ”add only” approach to the schema. This is to preserve the behavior of the API for existing clients. And this works because it is the clients who specify the structure of the response data in the queries. This means that as long as you can do with the ”add only” approach, GraphQL is very effective. If there is a need to make changes that will break the old versions a custom solution, as when using REST, has to be used for versioning. When experimenting with the need to break old features of the API we landed in having to create a new GraphQL schema at a new endpoint representing the new version of the API. It is also possible to break existing clients and then in whatever way seems fit notify them about the changes. With either of the solutions you would miss out on the fact that GraphQL is pull based. Pull based means that the server maintainers do not need to inform clients of the changes. Lee Byron, one of the creators of GraphQL, compares this to how columns work in SQL tables. A column may be added to a table without breaking old queries but make changes to an existing one and the old queries will break.

Caching

Caching is one of the major selling-points of REST. A REST API is able to leverage the HTTP caching, see section 2.1.5, without any extra work on the server or in the client. GraphQL is a stateless server side technique and is therefore not meant to provide caching. This is because server side caching would imply having a state. But as said with REST one will be able to leverage the HTTP caching with no more work than just setting a flag in the request header on the client side. But even if HTTP caching is used for a GraphQL API, its query nature does not leverage HTTP caching that well. When using GraphQL, the problem with over-fetching and multiple round trips is solved because the client specify what data it wants. This comes at the cost of giving up the possibility to leverage the HTTP caching in a good way at least if different queries are being used. A REST APIs gain from using HTTP’s caching depends on the implementation of the resources. If strict REST (maturity level 3) is followed and if the resources are small caching could possibly save a lot of unnecessary traffic.

For GraphQL there are also client side frameworks such as Relay, also developed by Facebook, that offer client side caching possibilities. Client side frameworks for handling caching are outside of the scope of this paper, but it should be mentioned that there are more possibilities to have client side caching with GraphQL. How well these frameworks perform, what constraints they put on your application or how much extra work they need have not been looked into.

Type System

REST can use the Content-Type header in the HTTP responses to describe the type of the data. This might be seen as a type system but in that case it is both very limited and weak. In comparison, GraphQL is strongly-typed, each query and the corresponding response may be evaluated to have correct data types. We would argue that HTTP Content-Type headers not provide enough to be seen as a type system. Content-Type Headers cannot help developers to validate their data in a secure manner, using GraphQL the client can be certain that the returned data is of the desired type. The difference between strong and weak typing is that a strongly typed language is more likely to generate an error or refuse to compile if the type is not matched.

5.2.4 Comparison of implementation technology

Implementation languages

As seen in the results a couple of implementations of GraphQL exist. Before one chooses to use an external framework in a large scale production project there are a lot of factors to be considered. The most important is probably the level of maintenance of the external framework. It gets extra important if the software used affects many parts of the project. It is important that the introduced dependencies are both well made from the beginning but also that one may rely on that eventual bugs will be solved and new features will be introduced when needed. Otherwise the developers might not only need to develop the main application but also frequently commit fixes to the sub-projects, which the main application depends on. Of course this is almost impossible to guarantee but by using active open-source projects with many contributors one may at least reduce the risk of such a situation. Thus we decided to set some restrictions on what implementations should be counted as viable alternatives to use. The projects need to implement the full GraphQL functionality and additionally they need to have at least five active contributors and have a relative continuous flow of commits to the project. That left us with five languages with potential implementations. Compare this to REST that may be implemented in a relatively simple way, without any dependencies to external projects, in every language that can handle HTTP-requests. This makes the choice of implementation language a less important choice when using REST. Except for efficiency and availability of developers it does not matter a lot, compared to GraphQL where the availability of a good GraphQL implementation is of the highest importance. One may of course use a do-it-yourself solution and implement GraphQL from scratch in whatever lan-

guage fits the project best. That would not only take a lot of time but would also result in the responsibility of maintaining the GraphQL-implementation in the future.

Client-side frameworks

REST and GraphQL run server-side and is not dependent of any client-side frameworks. Despite this one might want to use Relay when working towards a GraphQL server due to the relatively complex queries. In our test scripts we ended up with quite large GraphQL queries and found it easier to implement utility functions to create the queries. In applications with the need of even more interaction with the server creating the queries might be very tedious compared to interacting with a REST API.

Licenses

The specific licenses that software is released under greatly affect how it may or may-not be used in production software. The use of software released under some licenses may set constraints on the licensing of the other software, i.e. using GPL[33] licensed software in an application obliges the developers to release the whole application under the GPL license. This might not be a problem, but in some applications there are business secrets that require that the source code is kept hidden. Thus more "allowing" licenses as MIT or BSD might be preferable as they do not restrict the choice of licensing of the whole application. Almost all applications have to rely on external dependencies in some way or another, at least by language run-times and standard libraries, but to reduce the dependencies in terms of licensing leaves more flexibility for an optimal solution for the business. REST does not introduce any extra licenses as the whole architecture is only dependent on HTTP. This is a big difference compared to GraphQL that has to be noticed when choosing between any of the other.

Chapter 6

Verdicts

6.1 Verdicts

It has been a challenging process to derive what REST actually is since it is an architectural style proposed in a doctoral dissertation. This means that it is more flexible and abstract compared to GraphQL.

Even though both techniques offer a solution to the same problem and as can be seen in Table 4.1 they have a lot in common on a principal level. The actual decisions that have to be made on a conceptual level can be seen in Table 4.2 and they differ a lot. There are many more alternatives for each decision for a REST API compared to a GraphQL API. The flexibility of REST is more preferable than the single uniform solution offered by GraphQL in some cases but it would require less effort for developers to decide how to design a GraphQL API than a REST API. In Table 4.3 one can see that both GraphQL and REST require similar technology decisions to be made but the possible alternatives vary between the techniques. A designer of a GraphQL API would have the possibility to use another protocol than HTTP to transfer the messages between the client and the server. The REST designer would need to put more care into deciding which payload format to use.

A big difference can be seen in Table 4.4 of the decision model. These are the decisions of which implementation to use. When designing a GraphQL API the developer is up for both more decision and alternatives. It introduces risk and possible work to rely on external dependencies such as a GraphQL implementation. A developer of a REST API needs to make decision about which language to use but does not need to rely on external projects for the core functionality of the API. The developer could avoid all external dependencies if needed that makes it a solution that can be used for more types of applications than a GraphQL API. Additionally, performance has to be taken into consideration. For some applications such as the one that the use-case was based on, high performance gains can be made if choosing to use GraphQL. As our measurements shows, the network load may be reduced when using a GraphQL API that affects both performance for the end-user and the cost for the application provider.

Even if both techniques solve the same problem, e.g. communication in distributed loose coupled systems, they result in a very different process of decision making when actually realizing them. From our experience GraphQL would be excellent to work with when you control both back-end and front-end. When working with an API to be consumed by others we would choose REST because of its wide adoption and simplicity. A scenario where GraphQL is always preferred, is when REST results in a large amount of requests/responses and performance in terms of page-speed and network use is important. This might be on for example mobile applications. GraphQL outperforms REST when measuring latency and network traffic for the use-case specified in this paper, a use-case representing many of today's applications.

References

[1] M. Fowler, *Patterns of Enterprise Application Architecture*. Pearson, 2013.

[2] Apache. Date accessed: 16 May 2016. [Online]. Available: http://www.apache.org/

[3] Nginx. Date accessed: 16 May 2016. [Online]. Available: https://www.nginx.com/resources/wiki/

[4] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[5] C. S. Guynes and J. Windsor, "Revising client/server computing," *Journal of Buisness & Economics Research*, vol. 9(1), pp. 17–22, 2011.

[6] An introducction to software architecture. Date accessed: Feb. 2016. [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf

[7] (2010, 3) Richardson maturity model. Date accessed: 26 Jan 2016. [Online]. Available: http://martinfowler.com/articles/richardsonMaturityModel.html

[8] RESTful web services vs. "big" web services: Making the right architectural decision. Date accessed: Jan. 2016. [Online]. Available: http://www8.cs.umu.se/kurser/5DV095/HT09/literature/restvsbig.pdf

[9] (2015, 7) GraphQL in the age of REST APIs. Date accessed: 2 Feb 2016. [Online]. Available: https://medium.com/chute-engineering/graphql-in-the-age-of-rest-apis-b10f2bf09bba#.378467c50

[10] I. Grigorik, *High Performance Browser Networking*. O'Reilly, 2013.

[11] Spdy protocol. Date accessed: Feb. 2016. [Online]. Available: http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-0

[12] Http2 protocol. Date accessed: Feb. 2016. [Online]. Available: https://tools.ietf.org/html/rfc7540#section-5.1.1

[13] Http2 FAQ. Date accessed: Feb. 2016. [Online]. Available: https://http2.github.io/faq/#what-are-the-key-differences-to-http1

[14] (2011, 3) Analyzing web application performance. Date accessed: 25 Feb 2016. [Online]. Available: https://www.elie.net/blog/web/analyzing-web-application-performance

[15] P. Bansode. S. Barber. J.D. Meier, C. Farre and D. Rea. Performance testing guidance for web applications. Date accessed: Feb. 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb924375.aspx

[16] (2009, 9) Performance testing: Response vs. latency vs. throughput vs. load vs. scalability vs. stress vs. robustness. Date accessed: 26 Jan 2016. [Online]. Available: https://nirajrules.wordpress.com/2009/09/17/measuring-performance-response-vs-latency-vs-throughput-vs-load-vs_scalability-vs-stress-vs-robustness

[17] AWS cloud pricing principles. Date accessed: Apr. 2016. [Online]. Available: http://aws.amazon.com/pricing/

[18] Digitalocean pricing. Date accessed: Apr. 2016. [Online]. Available: https://www.digitalocean.com/pricing/

[19] (2015, 12) Virtual machine migration in cloud infrastructures: Problem formalization and policies proposal. Date accessed: 3 Apr 2016. [Online]. Available: http://www.lunduniversity.lu.se/lup/publication/7852890

[20] G. J. Popek and R. P. Goldberg, "Formal requirement for virtualizable third generation architectures," *Communications of the ACM*, vol. 17(7), p. 413, 1974.

[21] J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecture," *IEEE Software*, vol. 22(2), pp. 19–27, 2005.

[22] J. Klüster. F. Leymann. O. Zimmermann, T. Gschwind and N. Schuster. Reusable architectural decision models for enterprise application development. Date accessed: Feb. 2016. [Online]. Available: http://soadecisions.org/download/QOSA2007_4880_0015_0032.pdf

[23] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[24] (2016) www.docker.com. Date accessed: 12 June 2016. [Online]. Available: http://www.docker.com/

[25] (2016) www.nodejs.org. Date accessed: 12 June 2016. [Online]. Available: http://www.nodejs.org/

[26] (2016) www.expressjs.com. Date accessed: 12 June 2016. [Online]. Available: http://www.expressjs.com/

[27] Graphql. Date accessed: Feb. 2016. [Online]. Available: https://github.com/graphql/graphql-js

[28] Introduction to integration styles. Date accessed: Apr. 2016. [Online]. Available: http://www.enterpriseintegrationpatterns.com/patterns/messaging/IntegrationStylesIntro.html

[29] (2015) Why falcor? Date accessed: 6 May 2016. [Online]. Available: https://netflix.github.io/falcor/starter/why-falcor.html

[30] (2009, 6) Speed matters for google web search. Date accessed: 6 May 2016. [Online]. Available: http://services.google.com/fh/files/blogs/google_delayexp.pdf

[31] (2004, 8) Enterprise integration options. Date accessed: 29 Apr 2016. [Online]. Available: http://web.mit.edu/itag/eag-0.1/EnterpriseIntegrationOpts.pdf

[32] (2014, 2) Your API versioning is wrong, which is why I decided to do it 3 different wrong ways. Date accessed: 25 Apr 2016. [Online]. Available: https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/

[33] (2007, 6) Gnu general public license. Date accessed: 21 May 2016. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.txt