Communication Between iOS Mobile App and Backend

Trinh Tran

Haaga-Helia University of Applied Sciences
Degree Programme in Business Information Technology
Bachelor Thesis
2023

Author(s)

Trinh Tran

Degree

BITE Degree Programme in Business Information Technology

Report/Thesis Title

Communication Between iOS Mobile App and Backend

Number of pages and appendix pages

41 + 4

The growth in software engineering and booming emergence of mobile applications during the last a few decades has led to a fast development of backend and frontend system. Frontend is an extremely important exposure point between business and their customer, while backend handles the internal working, storing, and manipulating data. All software system caters to a flawless communication between backend and frontend, which is essential for ensuring the security and reliability of the overall system. However, despite its importance, there is currently limited research on this topic, particularly in the context of mobile application.

The theoretical framework of this thesis begins with an overview of the basic network protocols and components of client-server architecture, followed by a discussion of the challenges faced by this architecture. It then delves into the principles, advantages, and limitations of REST. Finally, the chapter explores the power and flexibility of GraphQL, examining its defining elements, building blocks, and best practices for designing high-quality APIs. The aim of this theoretical framework is to provide a comprehensive understanding of these architectures to lay the solid groundwork for the empirical research that follows in the subsequent chapters.

The findings of this thesis provide comprehensive understanding of REST and GraphQL, as well as compares these two popular communication styles. The thesis also examines F-secure SENSE and its use of both RESTful APIs and GraphQL, providing insights into the challenges faced when implementing both communication style in a single system. Overall, the finding of this thesis highlight the importance of selecting the appropriate communication style based on system's requirement, as well as benefits of combination of different styles to optimize system's performance and flexibility.

The thesis can contribute to developer community in selecting the appropriate communication style based on requirement of their system to optimize performance and flexibility. This research also introduces several research topics in the field of web-based API architectures and communication. A potential topic is to explore the best practices for integrating and implementing both RESTful API and GraphQL in a single system.

Key words

Backend, Frontend, REST, GraphQL, Server-Client Communication, iOS app

Table of contents

1. Introduction

Mobile phone is becoming an indispensable part in our life nowadays. Mobile application development has grown immensely to meet people's demand. This massive growth at the same time created a fierce competitive in mobile application market. When the application grows, it would need to serve a growing user base. The main goal of any application is aiming for a flawless performance. Communication backend and frontend system is vital to the software system's performance. This ceaseless back-and-forth cooperation between backend and frontend is required to enable data exchange for the app to function correctly. In a study titled "The Impact of Communication Latency on Distributed System Performance" (Korner at al., 2020) researchers analyzed the impact of communication latency on the performance of a distributed system. They found that latency had a significant impact on system performance, with longer latencies leading to slower response times and reduced throughput. The study highlights the importance of designing communication protocols that minimize latency.

The performance of the communication between backend and frontend has a significant impact on providing seamless user experience. In research about effects of system latency to UX (User Experience) and performance in VR (Virtual Reality) applications, researchers found that even even small delay or flaw in communication between backend and frontend, could significantly reduce user satisfaction and engagement (Caserman, Martinussen & Göbel, 2019).

Studies highlight the importance of great impact of communication between the backend and frontend in software system in different aspects: Data exchange, User Experience, Security, Scalability, Maintainability. The author's experience working closely to software product provides her to some extent the understanding about the influence of communication between backend and frontend to performance of a software system. This sparks an interest and was the initial motivation for the author to learn more about different frontend and backend communication styles for a comprehensive understanding about this topic.

1.1 Research objectives and research question

My research question focuses on examining different styles of communication between backend and frontend systems. By exploring this question, my research objectives to explore about the challenges and opportunities in designing and implementing secure and reliable communication between backend and iOS client.

In order to answer the primary research question. The following sub-questions were formulated and will be answered throughout the thesis:

1. What are the main components of a communication system between backend and frontend? What is client-server architecture, network protocol, API, components, and how those components cooperate and work together?
2. What are the most popular communication styles between backend and client, especially iOS mobile app. What are those style's characteristics, advantages, and limitations?
3. What are advantages and challenges faced when implementing both communication style in a single system in a real-life example iOS client app (F-secure SENSE)

Those research question provide a fundamental foundation upon which the entire study is built. It is the starting point of this thesis and guide the author's effort in conducting research and develop arguments throughout this thesis.

1.2 Why F-Secure SENSE?

F-secure is a Finnish global company operating in cybersecurity field. F-Secure is a successful player in cybersecurity when offers high quality products, which won customer's heart by their loyalties and enthusiasm toward their product and company, as well as numerable global honorable awards and measures (F-Secure, 2023).

F-Secure SENSE Connected Home Security is a cybersecurity product offer by F-Secure Cooperation, which will protect your home connected devices against cyber-attacks and threats. SENSE's functionalities are embedded in Service Provider Wireless Computer Network Routers or home gateways to enable easy uptake. F-Secure SENSE is a powerful mobile application that provide control and protection for all devices connecting to the home network by set of powerful features and functionalities namely Central Device Control, Threat Blocking and Visibility, Device Recognition and Profiling, Family Rules.

F-Secure SENSE is a global reliable product designed by highly skilled specialist. With their expertise, SENSE consumer apps consistently receive NPS scores of 40+. NPS (F-secure, 2023) is a global measure for customer loyalty, enthusiasm with a company calculated by asking customers questions "On a scale from 0 to 10, how likely are you to recommend this product/company to a friend or colleague?". Given the available -100 to +100 range, any > 0 score is read as "good" since it indicates the product/business obtain higher level of promoter than detractors. In 2018, top-notch companies namely Netflix had an NPS of 64, PayPal scores 63, follows by Amazon 54, Google 53 and Apple 49. Therefore, F-Secure SENSE which scores NPS

consistently 40+ is incontrovertible evidence for its quality and confidence in User Experience. (F-Secure, 2023).

F-Secure, a cybersecurity company that has been innovating in this field for over three decades, has designed this product to safeguard millions of individuals and tens of thousands of businesses against cyber threats. Most new connected devices are not designed with security in mind, and each device poses a potential threat to our digital privacy and security. However, F-Secure SENSE connected home security provides protection against ransomware, bots, and other online threats to ensure online privacy and security. The security feature also includes parental control features, which filter unsuitable content and set healthy boundaries for children's time spent online.

F-Secure SENSE connected home security offers several key features, including smart home security, browsing protection, tracking protection, botnet protection, family protection, and the ability to manage devices in your home network through the SENSE app. With SENSE, you will receive notifications if any devices start behaving oddly, and you can block internet access for those devices. You can explore the internet safely and perform banking and shopping activities without worry, thanks to SENSE's ability to block malicious or compromised sites. Additionally, SENSE blocks tracking sites from collecting data about you to ensure your privacy. Finally, the security feature offers protection against botnet attacks by blocking traffic from compromised devices to the attacker's command and control center. (Appendix 7)

Later in this study will attempt to analyze this success of F-Secure SENSE in winning customer’s heart in the technical aspect of how F-Secure SENSE handle communication and cooperation between client mobile application SENSE with SENSE backend. The aim of this study is to extract the most comprehensive, applicable insights on how a good Backend Frontend communication would look like to assist developers who aims for application’s great UX, which is ultimately translated into customer’s retention and loyalty.

1.3 Structure of the thesis

Chapter	Name	Main content
Chapter 1	Introduction	• Providing an overview of the research topic's context and background • Outlining the research objectives and questions, as well as the thesis's scope and structure.
Chapter 2	Theoretical background	• Providing theoretical framework to support the thesis.
Chapter 3	Methodology	• Discussing about the methodologies used in this thesis, explain the reason for selecting them, and their limitations.
Chapter 4	Results and discussion	• Presenting the findings • Concluding and answering the primary research question
Chapter 5	Summary	• Reviewing research question, main findings, contribution, limitations • Evaluate thesis’s objectives and personal learning
	Sources	• Listing the references used in the research
	Appendices	• Listing the charts, statistic, figures supporting the research

2 Theoretical Framework

The topics this theoretical chapter will discuss about Server-Client Architecture, APIs, REST architecture in detail, including its principles, advantages, limitations, and design considerations. Also, GraphQL is introduced as a powerful and flexible alternative to REST, with a detailed discussion of its elements, building schemas, queries and mutations, validation, and comparison with REST. Other communication styles between backend and mobile applications are briefly mentioned.

2.1 Server-Client Architecture

People are using computers networks for performing many tasks. Most business application requires communication with other internal and third-parties application to function these tasks. When the sheer volumes of users and user's request increase, a significant strain is caused to servers and network. Client-server architecture is introduced as a solution for this strain with significant emergence of requests. See Figure 1 for the overview about Client-Server Architecture.

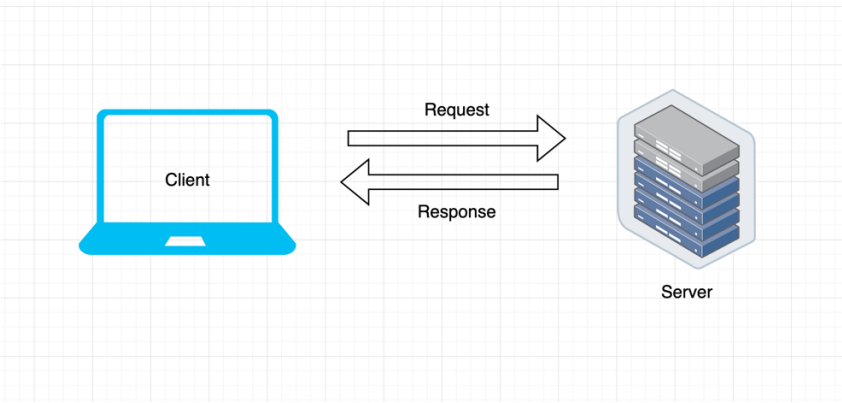


Figure 1. Overview about Client-Server Architecture.

2.1 Network protocol

"Sir Tim Berners-Lee produced the first version of the Hypertext Transfer Protocol (HTTP) as part of the Worldwide Web project started in 1990" (J. Cox, N. Jones, J. Szumski, 2012)

Berners-Lee's initial proposal included three main advancements: HTML, HTTP, and the URL. HTML specified a method for formatting text, HTTP specified a method for transmitting data

between a server and a client, and the URL specified a way to precisely identify a resource across a network of computers. (J. Cox, N. Jones, J. Szumski, 2012)
Network protocol conventionally established rules and standards to regulate how data is transmitted, formatted, received between different computer network devices (from server and servers to endpoints) regardless their difference in standards, infrastructures, and designs. Protocol suite is a group of cooperating network protocols. TCP/IP is a protocol suite, which is commonly used in client-server model for network interconnection between devices in a private computer network (intranet or extranet)
OSI Model, which is formally a backronym for Open System Interconnection, is a standard model, in which one or more network protocols govern activities at each layer during telecommunication exchange.

HTTP (Hypertext Transfer Protocol) is a protocol that run on top of TCP/IP suite of protocols. Through HTTP protocol, data can be exchanged and communicated between client-devices and servers over the internet.

2.2 Request/Response interaction

Request/Response is the most fundamental and popular client-server interaction pattern. It's default interaction that web services framework and related to API use. It is required when there is a client require an immediate response without any delay from server. Communication between client and server is consider synchronous because the server process as soon as it receives the request from the client and returns the response over the same connection strictly ordered and follow a specific sequence. When submit the request, the client cannot proceed with other task but will wait until the response for the request to be received from back end (R. Daigneau).

Client-server architecture or alternatively refer to the client-server model which is distribute application framework consist of client and servers; Client-server architecture breaks down the tasks between clients and servers that resides on the same system or connected via computer network. "The separation of concerns is the core theme of the Web's client-server constraints" (Rest API Design Rulebook M. Masse, Web Architecture, Introduction, not knowing pages). A clients or users are connected to the central server and send a request to the server whenever client need any service. The servers then process the request and send the response to the client or users.

This communication is enabled by HTTP protocol. The server exposes bunch of API that are accessible via HTTP Protocol. The client can directly call the services by sending HTTP request. That is where REST API comes into the picture.

2.3 Components of Client-Server Architecture

Server is a software that usually operates on a remote machine and can be accessed by a user's local computer, it receives and process requests from clients. Client is a computer software that runs on a user's local or remote computer and connect to the service. Client which will takes user's input and sends request to the server, are the software that request resources and services made available by the server.

As you can see in Figure 2 about the elements that make up the client-server architecture network-based intermediaries is the component that takes care of distributing incoming request across a group of servers to manage traffic and optimize resource usage (Load balancer). Network Protocol (TCP/IP) is also a part of network-based intermediaries which will distribute the application data into packets that can be transmitted across the network. Once connection is established in TCP protocol, the connection will be maintained until communication exchange between client and server is done. Whereas IP is a protocol in which data packet while being transmitted through the internet is unrelated to any other data packet.

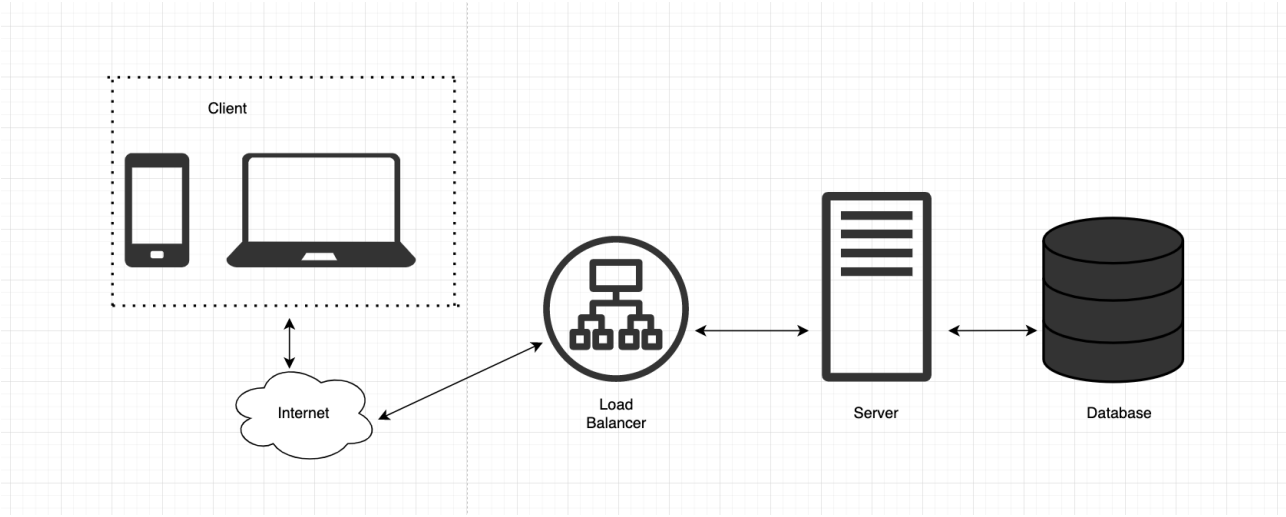


Figure 2. The elements that make up the client-server architecture

2.4 Client-Server Cooperation

In Client-Server Architecture, data flow is unidirectional, which forms a cycle. A cycle starts when the client sends a request to the server, the server will receive, process request, and respond with some sort of data back to the client via a protocol. Client cannot directly communicate to each other.

The flow will start when a client sends a request, the load balancer then routes the request to appropriate server. The server processes client requests an queries appropriate database for some data. Database returns the queried data back to server. The server eventually processes the data and sends back to the client.
Client-Server vs Peer-to-Peer Architecture.

While separation in task between Clients and Servers is the point of Client-Server Architecture. In Peer-to-Peer architecture, each computer is considered a node in the system, they are equally privileged and act as both suppliers and consumers and resources, such as network bandwidth and computer processing. Most interactions are directional.

2.5 Challenges of Client-Server Architecture.

The most obvious challenge of Client-Server Architecture is availability. With Client-Server model, the application will only work only if the server be always online and available to the clients. In case of any software, operation system or hardware failures and unavailability, it will turn down the application.

In Peer-to-Peer network, availability is on the other hand, solves this issue because each computer in the network is also a server. When a computer is facing failures, the peer-to-peer system will find the best clients and will request from them. This creates an advantage for application availability that doesn't demand the development any complicated high availability solution.

Second challenge for client-server architecture is high load or unexpected demand on the server. As a subset of availability issue, but it is more costly and more challenging to solve. For a Client-Server model to work properly, sufficient capacity at the server to satisfy high and unexpected demand at any time is required.

Scalability means growing with your application and it's one of the key issues with Client-Server Architecture. An application's data, especially, that of big enterprise's application is growing

enormously every day. Scaling infrastructure to handle growing immense amount of data means massive expense to increase server, storage, and network infrastructure.
Upsides of Client-Server Architecture

While Peer-to-Peer is decentralized node of computer system, Client-Server Architecture is decentralized network of systems with all data is amassed at one place, which will especially be beneficial to the network administrator since they have full control over management and administration. Whatever issue occurring in the entire computer network can be investigated and solve at one centralized point. And, also to this, the clients have the authority to access any file residing in the central storage at any time.
Client-Server network data is well protected due to its centralized architecture. Privilege to access the centralized storage is enforce with access control in the way that only authorized users with granted access will be able to access the specific data that they are given the access for. Therefore, data security is better protected.

Accessibility is one of the key upsides of Client-Server Network. Regardless location or the platform, every client is provided with the same accessibility to the system. Request allocation is done by a load balancer, therefore, tasks will be configured and performed in the most efficient route.

2.6 API

An Application Programming Interfaces (API) is a software interface that allows data exchange from two computer systems over computer network by sets of definitions and standards. It's sometimes referred as a contract between client server; API establishes a content call from the user and content response from the server. Diagram in Figure 3 describe API as a moderator to enable communication between user or client and web server or resources while maintaining

security, control, and authentication.

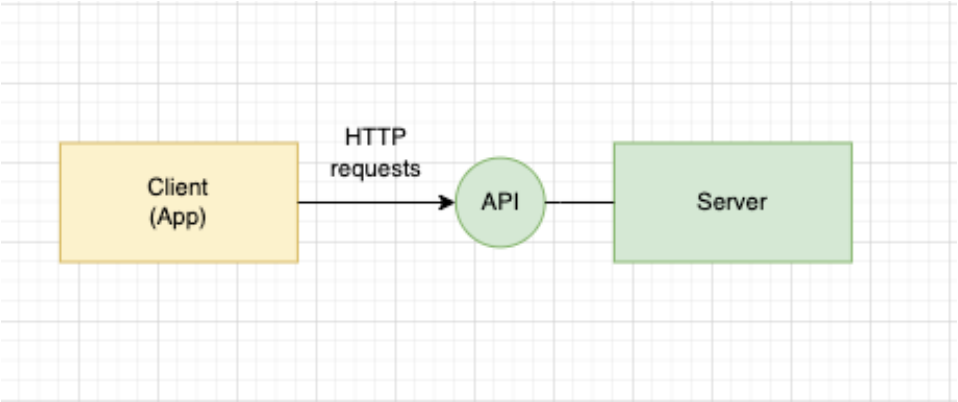


Figure 3. APIs support communication in client-server architecture.

2.6.1 The characteristics or components that define a high-quality API.

Handling communication between mobile application and backend requires developers to work a lot with APIs, therefore, it's crucial to know what define a good API to ease the development work, as well as the more efficient the server-side code will decide a lot on that of the client-side. According to Doglio F. (2018) there are a few factors to keep in mind when designing a good API. A comprehensive list of essential components that must be present to create a well-designed and effective Application Programming Interface (API), which is a set of guidelines, protocols, and tools used for building software and applications that interact with each other through a defined interface.

Developer friendly: Your system should be comprehensive to developer, and they when dealing with your system should not suffer to understand what those are and how they work.
Extensibility: Extensibility is the ability of adding new features without breaking clients
Documentation: Good and up-to-date documentation will help new developers to pick up your API quickly
Proper error handling: There should be always plan and preparation when things go wrong.
Provide multiple SDK/libraries: Simplicity of the system decides efficiency of it.
Security: Any global system needs to keep security of it in mind when designing APIs
Scalability: Any good API should be easy to scale up and down to properly provide its services.

Developer-friendly: Developer experience (or DX) is an aspect should be taken into consideration when designing a good API because the APIs will not be used by its designer but also by other developers. This means that your APIs should be as easy to use as possible to cater for a great DX, which translates into more developers and application using the APIs.

Portability: To make an API a good one, it is recommended to use a familiar network protocol which have portability in many existing technologies. There are several standards network protocol which is having libraries and modules available in many programming languages (e.g., HTTP, FSP, SSH, etc.)

In this paper, we will discuss under assumption that HTTP is the protocol chosen for REST. It is very popular protocol, which is supporting any modern programming language and create the fundamentals for our internet network. Therefore, we can assume that developers have basic understanding on how to use it, or at least they can easily learn how to use it.

That said, if custom protocol is more efficient in some use cases, and there are libraries to support plenty of existing languages, then it is up to designer to evaluate the best solution based on context of working.

Endpoint: is the contact point between client application and the API. A well-designed API is the one with easy-to-remember endpoints. Of course, the developers should always have documentation to read but the good endpoints are the one with mnemotechnic names which developer can understand their main purpose just by reading them.
For example:

GET /users/action1

action1 could be anything, the action name is unclear. The developers can't understand what the purpose of this endpoint is only by reading it. So, this is not a good endpoint.
A better example could be:

GET /users

This is a REST standard endpoint that listing items of this type. It provides developers more than enough information about the resources the request wants to access.
The endpoint is good to be easy-to-remember, but it easy more important to be consistent when being defined. A recommendation is to name endpoints based on the resources it handles instead of the actions taken. Here are some bad examples:

/getAllUsers
/submitNewUser

/updateUser
/getNumbersOfValidUsers

These APIs are clear enough to understand its purpose only by reading it and easy-to-remember. From the first glance they seem not bad, but they can be better. Each new item addition will cause extra endpoints. The APIs don't have a uniform in naming, so the application's developer will have no clue how these APIs was named. With the current naming scheme, there are several possibilities to name for supporting id images of the user:

/addNewIDImageToUser
/getUsersIDImage
/addUserIDImage
/listUsersIDImages

The list will go on to a really big list of endpoints, which increase complex for both server-side code and client-side code. It will reduce the simplicity of the system meaning the ease to work with. Solution for this is to generate an easy-to-use uniform interface across the APIs. As discussed in Uniform Interface constraints, REST suggest a resource-centric interface. With HTTP, you have verbs to indicate action of the request (See Table 1).

Old Style	**Rest Style**
/getAllUsers	GET /users
/submitNewUser	POST /users
/updateUser	PUT /users/:id
/getNumberOfValidUsers	GET books (the number cab easily be return with this endpoint)
/addImageIToUser	PUT /books/:id
/getBooksImages	GET /books/:id/images

Table 1. Comparison of Old Style APIs and REST APIs.

Another aspect of developer-friendly API is transport language used. XML has been used for years, but XML-JSON is gaining traction for its simplicity, human-readability, and portability for many datatype. XML-JSON is a great option, but it is not a silver-bullet for all issues. It's always better with options, and REST provides you with options. Since HTTP is the protocol that REST is

based on, developers can use content negotiation which is a mechanism allows clients to specify the supported format they want to receive. This allows more flexibility on the API.

Extensibility: There are some big APIs such as Google APIs, Facebook APIs or Twitter APIs that receive billions version if the three are still many clients are working with the old one. A good API designer will provide developers with version options to choose, keeping the old version long enough so that the clients have time to move to the new version. Otherwise, the clients which are using the old APIs will undoubtedly break.

Up-to-date documentation of calls a day. The migration could spell disaster if the original wasn't designed right. A good API with extensibility will takes this aspect into consideration:

- Is new endpoint easily added?
- Is version backward compatible?
- Is client still able to work with old version if new API version is released?
- Is updating new API version easy to client?

It is a bad move to totally shut down without any more support with an old version when migrating to a newer one.

Regardless how mnemotechnic the endpoints are, documentation should be always available for further learning and stay updated. A good API documentation is not limited with a few lines of explanation on how to access the endpoints, but all the information needed to know when using the endpoint (e.g., features, version, note, etc.) should be clarified. Some providers even provide developers with simple web UI to try out the API, which is friendly to the newcomers. A good example is Facebook's developer site which provide implementation and usage examples for all platforms that Facebook supports.

Proper Error Handling: A big part of developer's work is handling error and investigating on what is wrong. An application developer's life is much easier if they are working with APIs that provides exclusive and clear error handling documentation. Error handling can be translated to support developer on understanding what was wrong with their code and how to fix it.
There are two phases in an API life cycle that error handling needs to put into consideration:

Phase 1: Client Development phase

The developer during this phase will need to implement the required code to consume the API. There is a high chance that they will have error on the request (e.g., missing parameters, wrong endpoints, etc.), which need to be handled properly. Good error handling API would provide enough information on what was wrong and how to fix it.
The figure below shows what happens when an API doesn't add error handling in the client development stage. The error message might give the developers some sort of clue about what is wrong, but the issue here is it is not very friendly to newcomers because it provides excessive unnecessary information which might be very confusing. This will take lot of time to understand and hurt developer's experience when investigating the error.

The largest application programming interface (API) that exists. In other words, they are seeking information on the most extensive and expansive set of tools and protocols for building software applications that is currently available. In this figure, we can see how much simpler and more friendly error message compared to the first one, which a proper error response should look like. The error message has only three attributes, but they provide enough information about:

- Whether this response is an error message?
- State clearly the issue and how to fix them.
- Provide custom error code, which provide developer ability to automate action for this error in their code.

Phase 2: The API is consumed and being used by end users:

During this phase, developers such as wrong parameter, wrong endpoint is not likely happened anymore but there can be problems caused by data generated by the user. For example, the request requires user input can encounter with some input issue from end users. API should be able to handle any type of error in the input data. A good response is like what was describe in a good error response in phase one: whether the response is an error, the issue and solution, and error code to formalize the action in case error happens.

Multiple SDK/Libraries When choosing an API to use in your client, developer should keep in mind whether the API massively used in different technologies and platforms, which mean whether you have different means to consume the API in your services. Consider choosing the open-source libraries, because the open-source community is growing a thriving, where developers continuously maintain and improves the libraries. So, the popular and recommended open-source libraries by developers are worth considering.

the largest application programming interface (API) that exists. In other words, they are seeking information on the most extensive and expansive set of tools and protocols for building software applications that is currently available.
Facebook AIP SDKs for iOS, Android, JavaScript, PHP, Unity, Google Maps APIs provides SDK for iOS, Web, Android. Amazon provide SDKs for AWS Service in different platforms and languages for example PHP, Ruby, .NET, and iOS.

Security When choosing an API, there are two big security issue to deal with those are:

- *Authentication:* **Who** are accessing the API?
- *Authorization:* **What** they can access?

Regarding security in RESTful system, there are a few aspects to remember. RESTful systems are stateless: REST define server as stateless, which mean that storing user data in session after initial login is not recommended.

HTTPs: On RESTful system that based on HTTP, HTTPs is more recommended thanks for its security quality by encryption of the channel, making it's harder to read the data while travelling over the internet. Some widely used authentication scheme out there: Basic Auth with YSL, Digest Auth, OAuth, OAuth1.0a, and OAuth2.0.

Scalability: As scalability is usually underestimated when choosing API, simply because when building the application, it's quite challenging to predict the growth of the application, as well as estimate the traffic the APIs must handle in case of the application grows. But a good API is the one support growing traffic but still perform well with quality. This flexibility allow application not only to scale up but down the resources that are affected but also support fault tolerance and help developers maintain cleaner code base.

2.7 REST Architecture: Principles, Advantages, and Limitations

2.7.1 A Comprehensive Definition and Analysis

In the realm of technology, the word "service" has commonly been utilized to denote any software operation that performs a specific business function. “Web services provide the means to integrate disparate systems and expose reusable business functions over HTTP”. (R. Daigneau 2011).

Technologies nowadays has introduced multiple ways to implement a service as diverse as CORBA and DCOM, to newer frameworks developed for REST/WSDL can be used to create services.

“Representational State Transfer” (REST) is the name that Fielding gives to his Web architectural style in his Ph.D. dissertation. As to F. Dolglio stated in his book - *Manage and Understand the Full Capabilities of Successful REST Development* (2018), the keywork from REST definition is Style. because the one of the most important aspects of REST is that it’s
“…not a guideline, not a standard, nor anything that would imply that there are a set of hard rules to follow”.

The REST architectural style is universally applied to API design for modern web services. A “RESTful” web services is the web services having REST API. This is where API’s resource model consisting of an assembly of interlinked resources, comes in the picture.

A REST API (also alternatively known as RESTful API) is Application Programming Interface that allows secure telecommunication exchange between two computer systems over the internet. RESTful API supports this information exchanges because they follow network protocol set of standards and rules for secure communication while optimizing client-server architecture called HTTP Protocol.

2.7.2 Why REST Has Become the Preferred Architecture for Web-based APIs.

F. Dolglio also analyze the areas The idea of REST (Representational State Transfer) has improved in software development and web architecture. Some of the key areas of improvement are:

Performance: Efficient and simple communication style proposed by REST allows boosting performance for the system adopting it. REST achieves this by using a lightweight architecture that follows certain principles, such as using standard HTTP methods, representing resources using URIs, and using hypermedia links to navigate between resources. One way that REST improves performance is by reducing the overhead associated with network communication. Since REST uses standard HTTP methods, it eliminates the need for specialized protocols or middleware, which can add latency and increase complexity (Aurora Solutions 2023). This makes it easier for systems to communicate quickly and efficiently. In addition, RESTful systems can take advantage of caching to improve performance. By adding caching headers to HTTP responses, RESTful systems can reduce the number of requests sent over the network, which can significantly reduce

latency and improve overall performance. This is particularly effective for systems that frequently access the same resources, as it reduces the need to retrieve data from the server repeatedly (Restfulap.net 2023)

Scalability of component interaction: This is a key area which any distributed system needs to handle. The simple interaction proposed by REST handles this area greatly. the simplicity of RESTful systems makes them easy to implement and maintain. With fewer moving parts and a clear set of principles to follow, RESTful systems can be designed, implemented, and debugged more quickly and with fewer errors.These benefits make REST an attractive architecture for systems that require high performance and scalability.

Modifiability of components: The separation between the clients and servers in the idea of REST allows components modified independently of each other at minimum risk and cost. (Dolglio, 2018)

Portability: REST can be implemented and consumed by any technology and language - technology and language agnostic. One of the key benefits of REST is its technology and language agnostic nature, which means that it can be implemented and consumed by any technology and programming language without the need for specialized tools or protocols. This agnostic approach is possible because REST is based on standard HTTP methods, which are supported by virtually all modern programming languages and frameworks, and can work with various data formats, such as JSON, XML, and HTML (Macoveiciuc, 2020). As a result, REST can be used in a wide range of applications, including web services, mobile applications, and Internet of Things (IoT) devices, making it a highly flexible and versatile architecture for modern software development.

Reliability: Another way that REST improves performance is by supporting statelessness (Packt Publishing, 2019). This enable easy recover after failure is enabled by the statelessness constraint proposed by REST. By maintaining no state between requests, RESTful systems can handle a large number of simultaneous requests without incurring the overhead of managing state. This can significantly improve scalability and performance for systems that need to handle high volumes of traffic.

Visibility: REST improves visibility because the full state of a request is determined by a single request message.

Component centric design: REST is a component centric design which allow the system to adopt it to be easily recover from failure, because the other component of the system is not affected

entirely. Therefore, system stability is maintained. Adding up feature on demand is greatly easier thanks for interconnecting components, which also minimize the risk of scalability of a system. REST helps the system that adopt it to have better accessibility to wider audience thanks for its portability with a generic interface, which can be implemented by wider range of developers.

2.7.3 REST API Design

In this section, this paper will attempt to discuss the most common yet essential API design patterns, each of which address the RESTful constraints. These essential patterns should be accounted and ingrained as crucial for API designs and implementation patterns.

Statelessness: “Communication between client and server must be stateless” (Doglio 2018). To understand what statelessness is, let discuss what is application state. Application state is the information about client from previous request and stored in the server for incoming request.

Statelessness is a constraint proposed by REST, which requires each request from client to the server must contain all the information needed to process that request an no information can be stored at the server side for any future reference. This is called statelessness. The client must be responsible for storing and handling the request-related information.

Stateless is essential in RESTful API because it helps scalability of the system which adopt REST style architecture. Since server doesn’t store any information of the request, server can handle and process any concurrent requests by deploying APIs to multiple servers. (Packt Publishing 2019)

Cacheable: In REST style architecture, the response from server to the client are explicitly labeled as cacheable or non-cacheable (Wilhelmsen, Pautasso, Booth, Erl, Carlyle, Balasubramanian, 2012). Cacheable component helps to eliminate some interactions partially or completely over the network when the client passes through a cache component, which might reuse from the previous request.

In the discussion above we talked about the constraint where the request from the client in REST style architecture must be stateless. However, in this section why cacheable is constraint. Is there any conflict between these two constraints. There is no conflict between these constraints. By reducing the latency of some network interaction thanks for cacheable component, REST style architect reduces the latency during a series of interactions. This constraint is incorporated to balance out the disadvantage impact from Stateless constraint.

Uniform interface/ Uniform Contract: Uniform interface, which is alternatively referred as Uniform Contract is the key constraint to separate non-RESTful a RESTful API. It proposes a uniform way of interaction with a server regardless type of device or type of application (website/application). According to Fielding's dissertation, "all services and service consumers within a REST-compliant architecture must share a single, overarching technical interface." (Wilhelmsen, Pautasso, Booth, Erl, Carlyle, Balasubramanian, 2012)

Layered architecture: REST-based solution allows the system to use multiple architecture layers. Separation between the components of the layers, and interaction where each layer can only use the one below it as well as communicate with the one above it regardless of whether its consumer's communication along its delivery path.
This constraint of REST-style architecture allow system to use deploy the APIs one server A, and store data on server B an authenticate request in Server. This form of interaction allows system to work properly with the massive traffic in the web of webs, which simplifies distributed architecture and allows independent deployment and involvement of individual architectural layers. (Fielding, 2000)

Code on demand: In REST-based solution, this constraint is optional but originally intended to allow service consumer to support the execution of deferred service logic when service consumer can execute the service logic more efficiently and effectively. According to this, server can provide a portion of executable code to the client. For example, XML or JSON is a common static representation of resources to be sent from server to client. But instead of that executable code to support a part of your application can be sent. (Fielding, 2000)
All these constraints when being enforced in building web services will enables communication an interaction from them. If a service violates any of those constraint, if cannot strictly deferred as RESTful services.

2.7.4 The Foundation of Database Management and Web Development

CRUD is abbreviation for Create, Read, Update and Delete, which are the basic operations you can do with data. New data can be created, read, updated, or deleted. CRUD operation are the major functions are constantly used for interaction with database applications in communication with Web services, CRUD corresponds to the HTTP method on how a client interact with a web services.

CREATE: request is sent to API to submit a piece of data, what happen to the database underneath is it uses INSERT INTO command to insert a new row with new provided piece of data to database table.
READ: request is sent to API to list the data, which allow user to 'read' it as the name of the operation states. What happens to the database when a READ request is sent to the API is it use SELECT command to list the requested data.
UPDATE: This request is used when user want to change a record in database. That said, this operation is sent to modify a piece to data by performing UPDATE command to the target table and column to be update.
DELETE: This operation as it names states is to remove a piece of data out of database. In the database a similar DELETE command is perform when receiving this request.
HTTP Request

Also known as HTTP verbs, RESTful APIs allow applications to have all possible CRUD (create, retrieve, update, delete) operations. Since RESTful style system is built based on HTTP network protocol, it suggests using a specific HTTP method on a particular type of request. These methods indicate actions to be performed with the web services. Here are the main HTTP request methods, or verbs:

GET: This method's purpose is like READ function in CRUD, which is for requesting representations of a target resources specified in the request. A GET request only retrieves data and doesn't change the state of the resource.
POST: This method is comparable to CREATE operation in CRUD, of which purpose is to send data to the target resource, in essence to create a new resource. POST request can potentially change the resource state.
PUT: When client want to replace current representations of the target resource with new data, PUT resource is used. This method is in essence used to update or edit something in the database. A similar CRUD operation with this method is UPDATE.
DELETE: As the name state, this HTTP method is used for deleting a specified resource, which is like DELETE CRUD operation does.
(Pang, 2021)
Apart from the abovementioned main HTTP methods which corresponds to CRUD operations, there are a few more:

PATCH: When client want to make a partial modification, PATCH can be considered as a set of instructions for how client expect a resource to be modified.
HEAD: This request returns headers, which will be returned from GET request. For example,

HEAD request is used when client want to get the data of the file size of a large download without downloading it.
TRACE: A method is used for diagnostic purposes, for testing path to a resource.
OPTIONS: This method is used to list the available options for communication with a target resource.
(Fielding, 2000)

2.7.5 Exploring the Limitations of REST

RESTful APIs have become the standard for building web services due to their simplicity, flexibility, and widespread adoption. However, like any technology, they are not without their drawbacks. In this chapter, we will delve into the main disadvantages of RESTful APIs and their potential implications for developers and users. Understanding these limitations is essential for building robust, secure, and scalable APIs that meet the needs of both developers and end-users. From performance issues to security vulnerabilities, we will examine the challenges that RESTful APIs face and explore potential solutions to mitigate these problems. By the end of this chapter, you will have a better understanding of the limitations of RESTful APIs and how to overcome them in your own API development projects.

2.7.6 Lack of standardization

One of the main challenges with RESTful APIs is the lack of standardization across different implementations. While the REST architectural style provides a set of guiding principles for creating Web services, there is no official standard for how a RESTful API should be designed and implemented. This can lead to confusion and difficulties in integration when different APIs do not follow the same conventions. (Teuchert, 2022)

2.7.7 Tight coupling

RESTful APIs often rely on a tight coupling between the client and server, which can make it difficult to change or modify the underlying implementation without breaking the API. This can be a problem in situations where the server-side implementation needs to be updated or modified, as it may require changes to the client-side code as well. This can also make it difficult to implement new features or functionality, as any changes to the API may require corresponding changes to the client-side code. (Gilmore, 2019)

2.7.8 Over-fetching and under-fetching of data

Another potential drawback of RESTful APIs is the issue of over-fetching or under-fetching data. In some cases, the API may return more data than is needed by the client, which can lead to unnecessary network traffic and slower performance. On the other hand, the API may also not return enough data, requiring the client to make multiple requests to get all the information it needs. This can lead to additional network traffic and slower performance as well. (Yellavula, 2020)

2.7.9 Versioning

Versioning is another issue that can arise when using RESTful APIs. As the API evolves over time, new versions may be released with added or modified functionality. However, this can cause problems for clients that are using an older version of the API, as the changes may not be backwards compatible. This can require additional effort to support multiple versions of the API or may require the client to completely rewrite their code to use the new version of the API. (Juviler, 2022)

2.7.10 Security vulnerabilities

According to Gartner's 2017 report, RESTful APIs are susceptible to a variety of security threats. Cybercriminals can exploit vulnerabilities in the API's design and implementation to steal sensitive data, disrupt operations, and gain unauthorized access to systems and networks. For example, API injection attacks, such as SQL injection and cross-site scripting, can allow hackers to inject malicious code into the API's input and output data, compromising its integrity and confidentiality. Similarly, authentication and authorization issues, such as weak passwords and session hijacking, can enable attackers to bypass security controls and gain access to unauthorized resources. To prevent these types of attacks, proper security measures must be implemented, such as input validation, output encoding, and authentication protocols. Input validation ensures that all data received by the API is in the expected format and within defined parameters, while output encoding prevents malicious code from being injected into the API's output data. Finally, authentication protocols, such as OAuth and OpenID Connect, can ensure that only authorized users have access to the API's resources. By adopting these security measures, developers can protect the API and its users from security threats and ensure the confidentiality, integrity, and availability of sensitive data.

In conclusion, RESTful APIs can provide a simple and standardized way for systems to communicate with each other over the internet. However, there are also several potential drawbacks to using RESTful APIs, including lack of standardization, tight coupling, over-fetching, and under-fetching of data, versioning, and lack of security.

2.8 Exploring the Power and Flexibility of GraphQL

2.8.1 What is GraphQL?

GraphQL is developed by Facebook in 2012 and released as open-source query language in 2015. It is a query language but unlike other query language like SQL (Structured Query Language). It has become more and more popular in developer community since introduction as a modern alternative to RESTful APIs. And it is widely used by companies such as GitHub, Shopify, and Airbnb, etc., (Graphql.org, 2023). The success has been gained so far by GraphQL comes from great number of advantages that GraphQL has over its competitors. It enables clients to specify the data they need and servers to provide only that data. GraphQL is efficient and flexible, simple, and standardized style of backend and client communication, which allows for multiple requests in only one request.

2.8.2 The elements compose GraphQL's popularity.

One of the factors creating the success for GraphQL is its efficiency and flexibility. GraphQL is an extremely friendly and flexible way to wide variety of client platforms. The APIs created by GraphQL can be used in different client platforms such as web and mobile application. GraphQL can reduce the amount of data transmitted over the internet since it is a strong-typed system, which allows clients to explicitly specify the data they need, and servers only returns the piece of data requested. Additionally, GraphQL allows live updates to the client when there is change in the server through subscriptions. GraphQL schemas are also a very fault-tolerant easy to maintain over time as it allows type checking and validation. w. GraphQL's benefits basically resolve some of REST API downsides including more efficient data transfer, explicitly specify the requested data, and easy to maintain IPI overtime.

Strong type system: S. Buna stated in his book GraphQL in Action about core of GraphQL that GraphQL has a strongly typed shemas (S. Buna, 2021). A strong typed programming language is the one that predefine each type of data such as string, integer, character, hexadecimal, etc. in that language. In research carried out by N. Tomatis and others in the research named "May You Have a Strong (-Typed) Foundation", Why Strong Typed Language Do Matter (N. Tomatis, R. Brega, G. Rivera, R. Seigwart, 2004) proved that a strong typed-language create a strong fundamental to produce a reliable program. In GraphQL, this is totally right when it is a strong-typed language, which allows developers to define a schema that specifies the types of data available and the relationships between them. The fact that GraphQL is a strong typed language has improved the

reliability of the schemas written in it and lay a strong foundation for the quality of the communication with the Backend in production.

Efficient and flexible: When discussing about GraphQL's specification, Buna determined that GraphQL can be used with any programming language and any data source, making it highly flexible and adaptable. It also supports real-time updates, caching, and subscriptions, which can be useful for building highly interactive and responsive applications. GraphQL provides a flexible and powerful way for clients to request specific data from a server, and for the server to return only the data that is requested, rather than a fixed set of data. This allows for more efficient communication between client and server, as well as a better user experience, as the client can request only the data that it needs, rather than receiving a fixed payload of data that may include a large amount of unnecessary information. With GraphQL, a client can request any combination of data that is available in the API, and the server will return only the requested data. This can lead to significant performance improvements, as the client is not required to request and process a large amount of unnecessary data.

Simple and standardized: One of the main benefits of GraphQL is that the available data and the structure of the data is explicitly defined in a schema, which can be queried by clients to discover what data is available and how it can be accessed. This allows for more efficient development, as clients can discover and use the available data without needing to refer to external documentation.

In addition to these benefits, GraphQL is also recommended by developer community since supports the concept of "resolvers", which are the functions to retrieve and manipulate data. Resolvers is a powerful development tool to build API. This feature of GraphQL allows creation of complex data yet easy to maintain queries. Another powerful feature of is "mutations", which are a concept used to modify data on the server. This allows creation, modification, and deletion of data using GraphQL APIs.

2.8.3 Building schemas

Creating a good GraphQL schemas involves several steps, including define the schema types, creating scalar types, design queries/mutations, and finally schema validation. The schemas should be well-defined, strongly typed, and organized in a way that makes sense for the data being queried according to Porcello and Banks (2018).

Identify the schema type.

A good schema is a schema that define clearly what type of data a client can read and write, the schemas are strongly typed, which is a powerful development tool. The action which the client will take should be well and clearly defined in the schemas. For example:

- Get list of students in Programming 1 class.
- Get a specific student with student id.
- Log in the user.
- Book a study room for a logged in user.
- Cancel the booking for the previous user.

The first step is to identify the data types that the API will support. This can include object types, scalar types, and enumeration types. Defining object types: Object types define the fields and relationships between the data. Each object type should be well-defined and organized based on the data being queried. According to Suma (2021), “These type-based objects are designed to work together to help us create a schema”.

Creating scalar types

In GraphQL, apart from object data type, which represents complex structures of data and contain multiple fields, scalar types are also data type but technically different from object types. They are used to represent simple, atomic values such as strings, numbers, and boolean. GraphQL also allows creation of custom scalar if needed, this enhances the flexibility when creating schemas in GraphQL (Apollo 2023). This feature is extremely handy when the client needs to request some data that are not built-in scalar type. For example, a custom scalar type could be created to represent a date or a URL.

2.8.7 Defining queries and mutations

Operations that API support, which are the actions user want to interact with the backend will be defined in queries or mutations. Base on the data requested, the queries and mutations should be organized and defined. Also, relationship between types using fields should be established and defined. There are three type’s relationship such as One-to-One, One-to-Many, Many-to-Many. Each field should be clearly defined and should provide all the necessary information to query or mutate the data. (Porcello, Banks 2018)

2.8.8 Validating the schema.

In APIs development using GraphQL, there is an essential step which is validating the schema after the schema is define. In this step the schemas should be checked to ensure that it conforms to the GraphQL the API's specifications and requirement. GraphQL Schema Validator is a great tool offered by GraphQL to perform this validation. GraphQL Schema Validator is a tool can check the schema for syntax errors, unused definitions, and other potential issues. A powerful feature of this tool is it allow developer to visualize the representation of the schemas, which can help developer to understand better the structure and relationship of the schemas. Thanks for this step, the developer can ensure that the API are well-created and functional correctly. Also, the potential error can be detected and caught early in the development process. Thereby this step can ensure the quality of the API build and reduce the likelihood of error in production. (graphql.org, 2023)

2.9 REST and GraphQL Comparison.

While REST APIs and GraphQL have some differences, they also have several similarities that make them both useful tools for enabling communication and data exchange between systems over the internet. that allows for more efficient communication between client and server, as well as a better user experience. It is well-suited for APIs that expose complex, structured data and will be used by many clients with varying data needs. However, it requires a more complex server-side implementation and may require more development and maintenance effort than a traditional REST API.

2.9.1 Similarity between GraphQL and REST

Both REST APIs and GraphQL use HTTP and JSON: One of the main similarities between REST APIs and GraphQL is that they both use HTTP as the transport protocol for requests and responses (Fielding, 2000; GraphQL, n.d.). HTTP is a widely used protocol that enables communication between systems on the internet and provides a standardized way to send and receive data (Fielding, 2000). In addition, both REST APIs and GraphQL use JSON (JavaScript Object Notation) as the primary format for sending data (Fielding, 2000; GraphQL, n.d.). JSON is a lightweight data-interchange format that is easy to read and write and is supported by a wide variety of programming languages (JSON, n.d.). Using HTTP and JSON enables both REST APIs and GraphQL to be flexible and easy to use, as they can be accessed and consumed by a wide variety of clients.

Flexible and scalable web services. Another similarity between REST APIs and GraphQL is that they can both be used to create flexible and scalable web services that can be consumed by a

variety of clients. REST APIs are designed to be scalable and flexible by adhering to a set of constraints that enable them to be easily integrated into a variety of systems (Fielding, 2000). These constraints include the use of a uniform interface, the separation of concerns, and the use of a stateless communication protocol (Fielding, 2000). Similarly, GraphQL provides a flexible and efficient way for clients to request specific data from a server, which can make it easier to scale the API as the needs of the client change. GraphQL also has a strong-typed system that enables the server to validate queries and ensures that the client receives the data it expects (GraphQL, n.d.). Overall, the **ability** to create flexible and scalable web services is an important similarity between REST APIs and GraphQL, as it enables them to be used in a wide variety of applications and scenarios.
Similarity 3: Both REST APIs and GraphQL can be used to enable communication and data exchange between systems.

Data communication solution: Both REST and GraphQL provide a way for systems to communicate with each other and exchange data over the internet (Fielding, 2000; GraphQL, n.d.). REST APIs enable systems to communicate by providing a set of endpoints that allow the client to access and manipulate resources on the server (Fielding, 2000). The client sends HTTP requests to the server and the server responds with the requested data or performs the requested action. GraphQL, on the other hand, provides a flexible and efficient way for clients to request specific data from a server (GraphQL, n.d.). The client sends a GraphQL query or mutation to the server, specifying the data that it needs, and the server responds with the requested data.

Similarity in actions: In REST APIs, CRUD operations are typically implemented using HTTP methods such as POST, GET, PUT, and DELETE (Fielding, 2000). For example, a POST request might be used to create a new resource on the server, a GET request might be used to retrieve a resource, a PUT request might be used to update a resource, and a DELETE request might be used to delete a resource. In GraphQL, CRUD operations are typically implemented as "mutations," which are a type of operation that can modify data on the server (GraphQL, n.d.). Mutations are defined in the GraphQL schema and are executed using a POST request to the GraphQL endpoint. This mutation creates a new user with the specified name and email and returns the id, name, and email of the new user. Similarly, GraphQL mutations can be used to perform update and delete operations on data.

2.9.2 Fundamental differences between REST GraphQL

In order to understand the benefits and limitations of GraphQL, it is useful to compare it to

traditional REST APIs. The following example show how different REST endpoint than GraphQL one.

REST endpoint: GET /users?status=active&role=admin&sort_by=name

This API call retrieves a list of all users who have the status "active" and the role "admin", sorted by their name. This endpoint uses query parameters to filter and sort the data returned by the API. The query parameters in this example include "status", "role", and "sort_by", and their values are "active", "admin", and "name", respectively. The API would return a list of users that meet the specified criteria and sort them by their name in ascending order.
Figure 4 show an example of a GraphQL query which retrieves a list of users with the status "active", the role "admin", and sorts them by their name. The query includes the fields "name", "email", "role", and "status" for each user object returned by the API. In contrast to the REST endpoint, the GraphQL query allows the client to specify exactly what data they want to retrieve, rather than returning all fields for the users. Additionally, the query parameters are included as arguments within the GraphQL query, rather than as query parameters in the URL.

```
query {
  users(status: "active", role: "admin", sortBy: "name") {
    name
    email
    role
    status
  }
}
```

Figure 4. Example of a GraphQL query

Ganatra (2021) defined the fundamental difference between GraphQL and REST is their approach to data retrieval. RESTful APIs follow a resource-oriented architecture, where clients make requests for a specific resource, and the server returns that resource along with any related resources. The server defines endpoints for each resource, and the client must make separate requests to each endpoint to retrieve the data.

In contrast, GraphQL uses a query-based architecture, where clients can specify exactly what data they need, and the server responds with that data. Instead of defining endpoints for each resource, GraphQL defines a schema that describes the data available on the server. Clients can then use this schema to construct queries that retrieve only the necessary data. This fundamental difference in approach leads to several important distinctions between GraphQL and REST. With RESTful

APIs, clients may receive more data than they need, which can be inefficient, especially on mobile devices with limited bandwidth.

Additionally, RESTful APIs require clients to make multiple requests to retrieve related data, which can add latency and complexity to the client's code. GraphQL, on the other hand, allows clients to retrieve only the data they need in a single request, which can greatly improve performance and reduce the amount of data that needs to be transferred over the network. Additionally, GraphQL can provide more flexibility for clients, allowing them to easily retrieve related data and traverse the API graph in a more natural way. Table 2. summarizes the key differences between REST and GraphQL.

This query would retrieve the id and name for the user with id 123. The response would contain only the data that is specified in the query, rather than a fixed set of data. The Table 2. Show the fundamental differences between REST and GraphQL in from different angles.

	REST	GraphQL
Data fetching	Fixed set of data returned from predetermined endpoints	Allows client to request specific data from server
Multiple resources	Requires multiple requests to different endpoints	Allows client to request multiple resources in a single request
Evolving API	New functionality often requires changes to client-side code	Allows new fields to be added to the API's schema without breaking existing clients
Performance	May require multiple requests or receive unnecessary data	Allows client to request only the data it needs, improving performance
Caching	RESTful APIs can be cached at the server or client-side	GraphQL does not have built-in support for caching

Security	Vulnerable to injection attacks	GraphQL has its own set of security considerations

Table 2. GraphQL and REST difference in summary

2.10 Other Communication Styles between Backend and Mobile application.

REST is not the only answer when it comes to API questions although it is certainly a favorable option. There are a few alternatives for REST that don't include the REST standard.

SOAP: This is a valid alternative to REST for better discoverability compared to REST although it was introduced before REST. API discoverability in a nutshell means the ability to search and find the right API resources to complement and enhance a client's optimal performance (Rapid, 2022).

In SOAP's communication, XML messages are used to authenticate, authorize, and execute remote code. It tries to offer an interface for remote method communication. The downside of SOAP is its dependence on XML for its communication layer, which tends to be very verbose and not the best way to send the data across the wire. On the other side, SOAP solves the discoverability issue which is an issue system adopting REST encounters. HATEOAS which are meant to improve discoverability in REST are usually ignored by developers since it is optional and not trivial to implement. This causes a potential coupling between client and server. SOAP solves this as forcing discorability as a part of the protocol. In all the services publishing an API, there needs to be a WSDL file describing all the endpoints.

RPC: RCP is an abbreviation for Remote Procedure Call, where a client can just perform the calls like with a local dependency. While GraphQL's focus is on resources and data, RPC is more action centric. In RPC the transfer channel and protocol are completely hidden from the client and focus on the action, this is the key benefit that RPC is over than REST. Altextsoft (2020).

3. Research Methodology

3.1 Systematic approach

This thesis's methodology is based on a clear research question about the communication between backend and frontend systems in an iOS app that provides security for home devices. The thesis has followed a systematic approach to research by defining the research question, selecting appropriate research methods, collecting, and analyzing findings, and drawing conclusions. This included setting up a simulation setup to test the communication between backend and frontend systems and conducting an extensive literature review to evaluate the advantages and disadvantages of various communication methods. The research methods used in this thesis are appropriate and relevant to the research questions.

An experimental study has been carried out by setting up a simulation setup to test the communication between backend and frontend systems. Later in this research, the steps to setup the simulation environment for testing communication between SENSE iOS client and Backend will be explained in a more detailed manner (Chapter 4).

Additionally, an intensive literature review has been performed to evaluate the advantages and disadvantages of various communication methods, as well as increases immensely the creditability and reliability of this research. The research findings could be used to improve communication between backend and frontend systems. Discussion on the limitations of the current communication between Back end and iOS client implementation is potentially beneficial for the product in improvements in the future. The findings and potential impact of this research can greatly enhance the validity and reliability of this research.

A literature review is an essential part of this thesis as it provides an overview of the existing research and helps establish the research gap that the thesis intends to fill. The research as has carried out following Haaga-Helia University of Applied Sciences' standard guidelines for literature review.

Identify the research question: The first step was to clearly define the research question and scope of the study. As discussed in the previous chapter, the objectives of this research were well identified and stated. All the literature reference taken into this research is on the purpose to support the understanding advantages and disadvantages of various communication methods for a secure and reliable between backend and iOS client.

Search for relevant literature: Once research question is identified, researcher conducted a comprehensive search for relevant literature. Use academic databases such as Google Scholar, Oreilly.com, HH Finna, Theseus to find public print material, e-material including peer-reviewed

articles, books, and other relevant sources. Researcher has done a careful evaluation and filtrations for the sources. Being fully aware that not all sources are created equal, and the researcher considered the importance to critically evaluate the quality and relevance of the sources found. Look for sources that are reputable, published in peer-reviewed journals or books, and authored by experts in the field. Especially, the time of publication is an important factor to make sure that the reference sources are up to date and still hold great values to the topic at the time of writing. From this point of view, almost all the chosen references were published later than 2017.

It's important to avoid bias when selecting sources for your literature review. In this research, a variety of sources from different authors and institutions to ensure a diverse range of perspectives.

Once the relevant literature has identified and evaluated, the sources were organized into themes or categories based on the research question. The purpose of this method is to identify the key findings and research gaps in the field.

By following the guidelines above and critically evaluating the sources were found, this thesis aims to ensure that literature review methodology is a reliable source of information for this research.

3.2 Limitation and discussion

Experimental methodology is a powerful research method with the advantages of allowing researcher to establish practical experiments and study, provides a high level of control over the research environment, and allows researcher to replicate the experiment to test the reliability and validity of their findings.

In the case of this thesis, the use of experimental methodology allows researcher to set up a simulation setup to test the communication between the backend and frontend of SENSE iOS app. By manipulating variables in a controlled environment, researcher was able to observe the effects of different communication methods on the overall performance of the app, thereby draw meaningful conclusions about the advantages and disadvantages of different communication methods and provided practical insights that could be used to improve communication between backend and frontend systems. Overall, the use of experimental methodology in this thesis was highly suitable for studying communication between the backend and frontend of a network security services such as a router-backend-mobile app system.

While experimental methodology has many advantages, there are also some limitations to consider. One limitation is that experimental studies are often conducted in a controlled environment, which may not accurately reflect real-world conditions. In the case of this thesis, the simulation setup may not fully replicate all the complexities of a real production network

environment, which might include integration between a few more third parties such as router vendor, operator, and so forth.

Additionally, there may be other factors that could affect the communication between backend and frontend systems that were not accounted for in the simulation setup or the literature review. Finally, it is also important to consider the limitations of the literature review methodology. For example, there may be biases in the studies or sources that were included in the review, or there may be gaps in the research that were not identified. Overall, it is important to acknowledge the limitations of this research methodology and to carefully consider the implications of the findings considering these limitations.

4. F-secure SENSE and its communication with Backend

4.1 Set-up the environment

To explore the communication between SENSE app client and the backend, a simulation setup it was setup to assist the research. Once installed, all internet-connected devices in your home, including smartphones, computers, smart TVs, gaming consoles, and baby monitors the security feature of SENSE are protected against cyber-attacks.

The compulsory hardware to have SENSE set required for this setup includes: 1 SanDisk 32GB SD card, 1Raspberry Pi 4B case (Black), 1 Raspberry Pi 4B power adaptor, 1Raspberry Pi 4B board (2GB), 1 wired ethernet cable. Figure 5 show how to configure the hardware to have a setup that has simulated SENSE integrated router (Raspberry pi), and this router can provide internet connectivity to the iPhone device that install SENSE app.

To simulate the backend and frontend communication, hardware requirements include a Raspberry Pi is configure with SENSE image to become a router integrated with SENSE features. With the support of XCode, which is an Apple integrated development environment, SENSE iOS app was run in XCode in an iPhone mobile iPhone. SENSE features including smart home security, browsing protection, tracking protection, botnet protection, family protection is used, and the code is observed lively in Xcode to study the code that enables and support communication between backend and iOS application features.

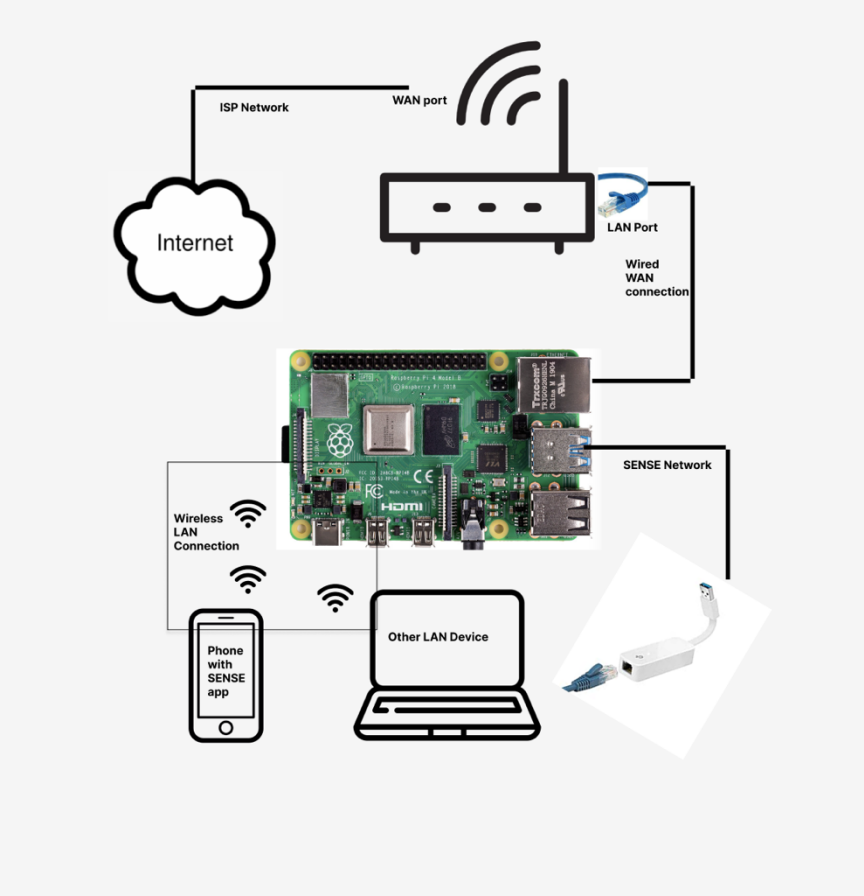


Figure 5. Hardware settings in simulation testing setup.

To integrate SENSE features to the Raspberry Pi to become a SENSE router. A Raspberry Pi was set-up to be a router by flashing SENSE image into it. There are several reasons why a Raspberry Pi might be used when creating a simulation system. Firstly, Raspberry Pi is a low-cost, compact and energy-efficient computer board that is easy to set up and use, making it an attractive option for hobbyists and developers looking to build a simulation system on a budget. Secondly, the Raspberry Pi's small form factor makes it ideal compact and integrated simulation experience. Overall, the Raspberry Pi is a powerful and versatile platform for creating simulation system.

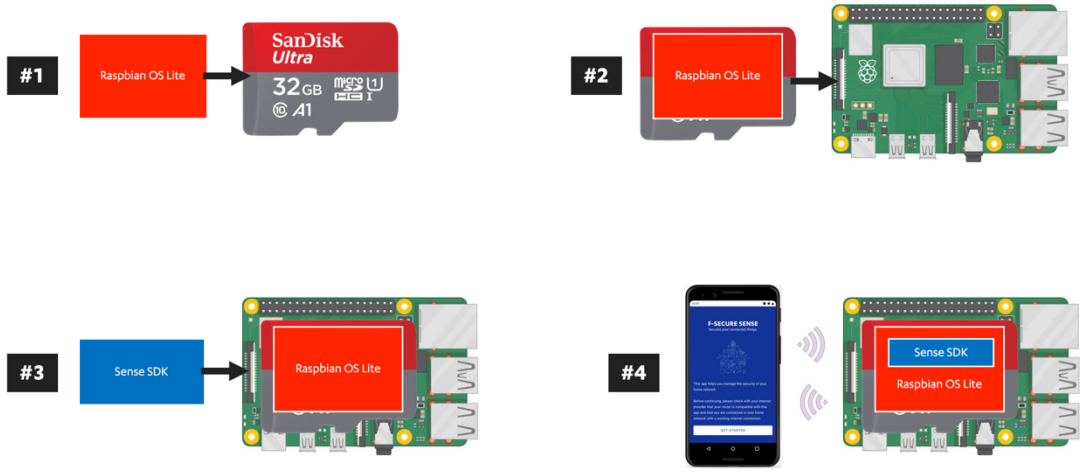


Figure 6. How to set up simulation environment to test communication between SENSE app and Backend with Raspberry Pi.

Figure 6 shows the steps were used to set up the simulation testing environment to test communication between SENSE app and Backend with Raspberry Pi. After obtaining the Raspberry Pi image, which is a software package management system. The image then needs to be flashed onto an SD card. After the Raspberry Pi is booted up, it needs to be connected to the internet via LAN socket and accessed via SSH using the provided username and password provided by F-Secure.

Once SENSE properties can be configured. Raspberry pi with SENSE integrated feature will work like a router and be able to provide the internet connectivity. Logs can be collected from the Sense docker, which can be useful in diagnosing any issues that may arise during setup or operation. At this step, an iOS phone device can connect to the SENSE network using the Network SSID and password configured in the previous step. Once joining the network, the application is run straight from XCode. After the installation, the app is ready to be pair to the network to enable the connection between the client mobile application and the router. Username and password from a license provided by F-secure was required for this step. After pairing succeeds, the testing and observation with the communication between the mobile application with backend started.

4.2 RESTful APIs in F-secure SENSE

As SENSE iOS application for administering Wi-Fi networks is using RESTful API, GraphQL, and WebSockets for different purposes in communication with the backend and SDK inside the router. RESTful API is used for CRUD (create, read, update, delete) operations on resources, such as devices, user accounts, and network settings. RESTful API provides a standard way for to interact

with the server, using HTTP verbs (GET, POST, PUT, DELETE) and resource URIs. This allows SENSE to easily manage resources on the server, such as blocking internet access for specific devices or updating network settings, and authentication.

F-secure SENSE is using reliable frameworks, which simplifies communication, service discovery, and authentication for client applications. It is designed to help developers easily connect their applications with different services in a secure manner. To use this framework, developers need to create an instance of its API class and set the desired endpoint (CI, Staging, or Production) and the authentication type. If using CCR credentials, developers must also set the CCR username and password. After setting the necessary information, developers can call the get APIs method with a list of services they want to access.

F-secure SENSE authentication use RESTful APIs because of advantages RESTful architecture offers. It is a popular and widely used approach for designing web services. RESTful APIs allow developers to expose web services using standard HTTP requests, which can be accessed by a variety of clients, including mobile applications.

One advantage of using RESTful APIs for authentication is that they are stateless, meaning that authentication can be performed on a per-request basis without the need for a server-side session. This can make authentication more scalable and easier to manage. RESTful APIs also allow developers to design APIs that are resource-oriented, meaning that each resource can be accessed and manipulated using standard HTTP methods such as GET, POST, PUT, and DELETE. This can make it easier to design and implement authentication mechanisms that are consistent and intuitive for developers to use.

Overall, using RESTful APIs in SENSE provides a scalable and efficient approach for managing settings modification and authentication, while also providing a standardized approach that can be easily integrated into a wide variety of client applications.
Example of RESTful API endpoint in SENSE app:

path: "/activation-status", method: .GET, scheme: .http

4.3. GraphQL in F-secure SENSE

GraphQL is typically used for more complex data queries and mutations. GraphQL provides a more flexible and efficient way for clients to request specific data from the server, by allowing

clients to specify the fields they need and reducing the amount of data sent over the network. This is particularly useful when dealing with large datasets or complex relationships between data.

The F-secure SENSE iOS application utilizes GraphQL extensively across various features spanning different aspects, from the router to the device and profile as well as event and settings. GraphQL provide a simple way to retrieve the router's settings, update them with parameters as well as query and mutate device and profile settings and details. Additionally, GraphQL enables clients to execute more complex operation efficiently from multiple sources or make multiple requests to a server, facilitating functions such as deleting devices, linking devices to a profile, and accessing a wide range of features.

SENSE iOS mobile application utilizes the third-party package to manage the client-side cache and create a network layer to connect to a GraphQL server easily. This package is a popular GraphQL client that has been widely adopted by the developer community due to its ease of use and flexibility. By using this third-party package, SENSE can efficiently manage the client-side cache and optimize network requests, reducing the amount of data that needs to be transferred between the client and server. It provides a flexible caching mechanism that allows developers to define how data is stored and retrieved from the cache. This means that SENSE can optimize the cache based on the specific needs of the application, resulting in faster and more efficient performance.

Moreover, an advantage of this package is a network layer that simplifies the process of connecting to a GraphQL server. The network layer handles all the low-level networking details, such as HTTP requests, authentication, and error handling. This means that developers can focus on building the application's functionality without worrying about the complexities of network communication. See example of graphql query in SENSE app at Appendix 6.

This GraphQL query retrieves a list of devices associated with a specific router, along with various details about each device. The query is parameterized with a required routerId variable, which is expected to be a non-null string value. The query requests information about each device's smartHomeDevice status, detectedManufacturer, detectedModel, detectedName, detectedOs, detectedOsVersion, detectedType, deviceId, displayName, firstSeen, ipAddresses, lastSeen, mac, profileUuid, eppDetected, and enforcement. These details provide useful information about the device’s types, manufacturers, models, operating systems, and more.

This query is a good candidate for GraphQL over REST because it involves complex data requirements that may require multiple REST requests to fetch all the required information. With GraphQL, it is possible to retrieve all the information in a single request by specifying the required

fields in the query. Furthermore, GraphQL allows the client to specify exactly what data is required and the server only returns the requested data, reducing the amount of data transferred over the network. In contrast, with REST, the server typically returns a fixed set of data, and the client must parse and filter out the unnecessary data. Another advantage of using GraphQL over REST for this query is that GraphQL provides a strongly typed schema, which allows for better tooling and error checking during development. With REST, the data model is often less formalized, which can make it difficult to detect errors and maintain consistency in the API.

4.4. Why SENSE is using both GraphQL and RESTful API

F-secure SENSE is using both GraphQL and RESTful APIs, it suggests that the project is taking advantage of the strengths of each API design style to accomplish different goals.
RESTful APIs are typically used for resource-based interactions, where data is requested from or sent to a server via well-defined URLs and HTTP verbs. RESTful APIs are often used to expose specific functionality and data from a server and are frequently used in web applications for tasks such as authentication, CRUD operations, and searching.

GraphQL, on the other hand, is often used for complex data fetching scenarios where a client needs to fetch data from multiple sources or make multiple requests to a server to obtain the required data. With GraphQL, clients can specify the exact data they need, and the server will only return the requested data, minimizing the amount of data sent over the network and improving performance.

By using both GraphQL and RESTful APIs, F-secure SENSE can take advantage of the strengths of each approach. RESTful APIs can be used for well-defined interactions that involve simple data retrieval or modification, while GraphQL can be used for complex data-fetching scenarios that involve multiple sources or complex data transformations.
It's worth noting that using both APIs in a single project can add complexity to the project architecture and development process, as developers need to understand the differences between the two APIs and how to best leverage each one in different resources.

4.5. Challenges When Implementing Both REST and GraphQL in the SENSE

Using GraphQL in a mobile application is a great way to take the most use of both methods. REST is an established standard for authentication and authorization, and its use enables compatibility

with a wide range of authentication providers and libraries. Meanwhile, using GraphQL to query complex data requests enables more efficient and flexible data retrieval that reduces the number of API requests required and provides more control over the returned data.

However, there are also areas where improvements can be made. For example, using REST and GraphQL in the same application can lead to a more complex architecture and require additional maintenance. In addition, using two different technologies for different parts of the application may also require more specialized knowledge from developers, which can increase the learning curve for new team members.

Another consideration is that using GraphQL for querying complex data requirements may also require additional attention to security, as the flexibility of GraphQL queries can make it easier for attackers to craft malicious requests that access sensitive data. Therefore, proper measures need to be taken to secure the GraphQL API.

Overall, using REST for authentication and GraphQL for querying complex data requirements is a reasonable decision, but there may be tradeoffs and challenges to consider. One potential improvement could be to evaluate whether it is necessary to use both technologies or if a unified approach using either REST or GraphQL would be sufficient for the application's requirements.

5 Conclusion

This thesis examines different styles of communication between backend and frontend systems. The research objective is to explore a variety of challenges and opportunities in designing and implementing secure and reliable communication between the backend and the iOS client by exploring this question.

This thesis' methodology is based on a clear research question about the communication between backend and frontend systems in an iOS app. The thesis follows a systematic approach to research by defining the research question, selecting appropriate research methods, collecting, and analyzing findings, and drawing conclusions. This included setting up a simulation setup to test the communication between backend and frontend systems. It also included conducting an extensive literature review to evaluate the advantages and disadvantages of various communication methods. The research methods used in this thesis are appropriate and relevant to the research questions.

This thesis provides comprehensive understanding of REST and GraphQL, as well as compares these two popular communication styles. The thesis also examines F-secure SENSE and its use of both RESTful APIs and GraphQL, providing insights into the challenges faced when implementing both communication style in a single system. Overall, the results of this thesis illustrate the importance of choosing the appropriate communication style based on the system's requirements and the benefits of combining different styles for the purpose of optimizing the performance and flexibility of the system.

The author has gained a lot of new knowledge of different styles of communication between backend and frontend systems. She has also had a closer look into real-life successful project to enrich her knowledge in this explicitly topic. In this project, various academic research methods were discovered, and those would help the author in future higher education. Additionally, conducting the research project within a very short timeline has helped the author develop time management skill. Thanks to this research, the author had chance to discover different angles of the F-Secure SENSE app, so this will help the author a lot in the future professional career development.

Sources

C. Korner, C. Gavriluta, F. P. Andrén, M. Meisel, & T. Sauter. (2020). The Impact of Communication Latency on Distributed System Performance. URL: https://www.researchgate.net/publication/346454271_Impact_of_communication_latency_on_distributed_optimal_power_flow_performance. Accessed 3 May 2023.

P. Caserman, M. Martinussen, S. Göbel (2019). URL: https://www.researchgate.net/publication/337108947_Effects_of_End-to-end_Latency_on_User_Experience_and_Performance_in_Immersive_Virtual_Reality_Applications . Accessed: 3 May 2023.

R. T. Fielding, 2000. Architectural Styles and the Design of Network-based Software Architectures. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Accessed: 22 December 2022.

J. Cox, N. Jones, J. Szumski, 2012. Professional iOS Network Programming: Connecting the Enterprise to the iPhone and iPad. URL: https://learning.oreilly.com/library/view/professional-ios-network/9781118417164/. Accessed: 28 December 2022.

F. Doglio, 2018. REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development. URL: https://learning.oreilly.com/library/view/rest-api-development/9781484237151/. Accessed: 15 Jan 2017.

R. Greenlaw, E. M. Hepp, 2003. Client-Server Model. URL: https://www.sciencedirect.com/topics/computer-science/client-server-model#:~:text=The%20three%20major%20components%20in,application%20logic%2C%20and%20data%20storage.

R. Daigneau, 2011. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. URL: https://learning.oreilly.com/library/view/service-design-patterns/9780321669636/. Accessed: 20 December 2022.

M. Masse, 2011. REST API Design Rulebook. Available at: https://learning.oreilly.com/library/view/rest-api-design/9781449317904/. Accessed: 8 Jan 2023.
F-secure, 2023. F-secure SENSE – Connected Home Security. URL: https://www.f-secure.com/en/partners/operators/solutions/connected-home-security. Accessed: 23 Jan 2023.

M. Biehl, 2015. Architectural Style for APIs: SOAP, REST and RPC. URL: https://medium.com/api-university/architectural-styles-for-apis-soap-rest-and-rpc-9f040aa270fa, Accessed: 3 Jan 2023.
F. Doglio, 2021. Not All Microservices Need to Be REST — 3 Alternatives to the Classic. URL: https://blog.bitsrc.io/not-all-microservices-need-to-be-rest-3-alternatives-to-the-classic-41cedbf1a907. Accessed: 24 Jan 2023.

R. Perkins, 2021. HTTP Request Methods and How They are Written. URL: https://randyperkins2k.medium.com/http-request-methods-and-how-they-are-written-d9333510bfc7. Accessed: 15 December 2022.

Aurora Solutions, 2019. “REST API Performance Tuning: Getting Started”. URL:https://medium.com/rpdstartup/rest-api-performance-tuning-getting-started-7a6efefa9e20. Accessed: 12 March 2023.

Restfulapi.net, 2023. “What is REST”. URL: https://restfulapi.net/caching/. Accessed: 12 March 2023.
Restfulapi.net, 2023. “What is REST”. URL: https://restfulapi.net/statelessness. Accessed: 12 March 2023.

N. Yellavula, 2020, 2019. Hands-On RESTful API Design Patterns and Best Practices. URL: https://learning.oreilly.com/library/view/hands-on-restful-api/9781788992664/fff729a2-d426-4c0f-96a1-0c2a181941de.xhtml. Accessed January
2022

Macoveiciuc, 2020. “Beginner’s Guide to APIs, Protocols and Formats”. URL: https://frontend-digest.com/beginners-guide-to-apis-protocols-and-data-formats-f80cf7f30425. Accessed 8 March 2023

H. Wilhelmsen, C. Pautasso, D. Booth, T. Erl, B. Carlyle, R. Balasubramanian, 2012. SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST .URL: https://learning.oreilly.com/library/view/soa-with-rest/9780132869904/ch05.html. Accessed: 12 December 2022

A. Pang, 2021. “CRUD Operations Explained”. URL: https://medium.com/geekculture/crud-operations-explained-2a44096e9c88. Accessed, 3 January 2022.

N. Yellavula, 2020. URL: https://learning.oreilly.com/library/view/hands-on-restful-web/9781838643577/becffd48-bca9-4c3a-9d89-0a490c5dce49.xhtml. Accessed 8 January 2022.
J. Gilmore, 2019. “What is Loose Coupling in REST APIs”. URL: https://blog.dreamfactory.com/the-importance-of-loose-coupling-in-rest-api-design/. Accessed 8 January 2022.

J. Juviler, 2022. “API Versioning: A Marketer’s Guide”. URL: https://blog.hubspot.com/website/api-versioning. Accessed 14 December 2022.

D. Teuchert, 2022. “The 6 Biggest Challenges of REST API Testing”. URL: https://www.code-intelligence.com/blog/challenges-rest-api-testing. Accessed 12 March 2023

Graphql.org, 2023. “GraphQL Foundation” . URL: https://graphql.org/foundation/. Accessed: 10 March 2023

K. Doyle, 2020. “A head-to-head GraphQL vs REST performance faceoff”. URL: https://www.techtarget.com/searchapparchitecture/tip/When-GraphQL-wins-in-a-GraphQL-vs-REST-performance-comparison. Accessed 9 March 2023.
Amazon Web Service (AWS), 2023. “Designing your schema”. URL: https://docs.aws.amazon.com/appsync/latest/devguide/designing-your-schema.html. Accessed 7 March 2023

S. Buna (2021). “GraphQL in Action”. URL: https://learning.oreilly.com/library/view/graphql-in-action/9781617295683/. Accessed: 6 March 2023.
E. Porcello, A. Banks, 2018. “Learning GraphQL”. URL: https://learning.oreilly.com/library/view/learning-graphql/9781492030706/ch04.html#many-to-many-connections. Accessed: 7 March 2023

R. Ganatra (2021). “GraphQL Vs. REST APIs”. URL: https://hygraph.com/blog/graphql-vs-rest-apis. Accessed 12 March 2023

Altexsoft, 2020. “Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC”. URL: Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC. Accessed: Accessed 12 March 2023

Apollo, 2023. “Custom scalars”. URL: https://www.apollographql.com/docs/apollo-server/schema/custom-scalars/. Accessed 13 March 2023

Appendices

Appendix 1. Bad Error Handling Message Example


```
{
    error: "TypeError: Cannot read property 'query' of undefined at BooksHdlr.index
    (/home/fernando/workspace/github/api-design/./handlers/books.js:20:22) at
    ProcessingChain.runChain (/home/fernando/workspace/github/api-
    design/node_modules/vatican/lib/processingChain.js:76:13) at
    /home/fernando/workspace/github/api-
    design/node_modules/vatican/lib/vatican.js:162:36 at
    /home/fernando/workspace/github/api-
    design/node_modules/vatican/lib/defaultRequestParser.js:63:13 at
```

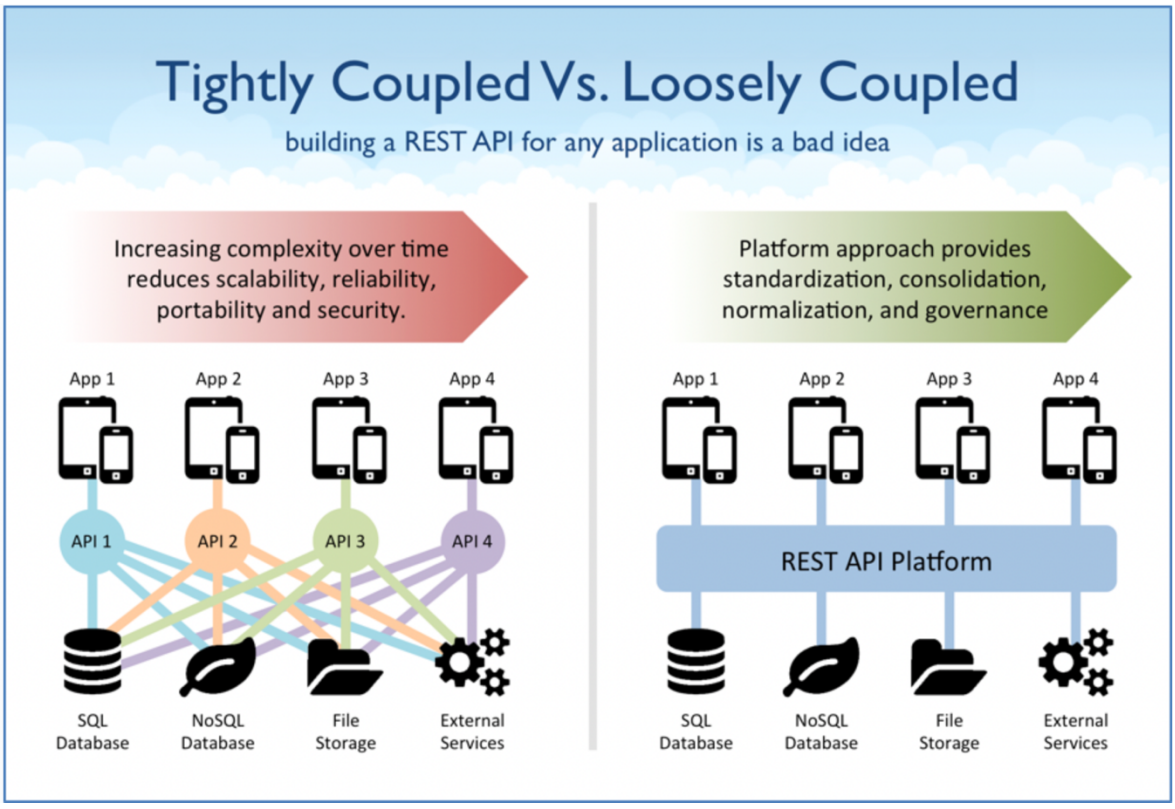

(M. Masse, 2011)


```
{
    error: true,
    error_msg: "Missing parameter "key" from querystring, please add it with the value
    you got from the development console on our site.",
    error_code: 4
}
```


Appendix 2. Good Error Handling Message Example

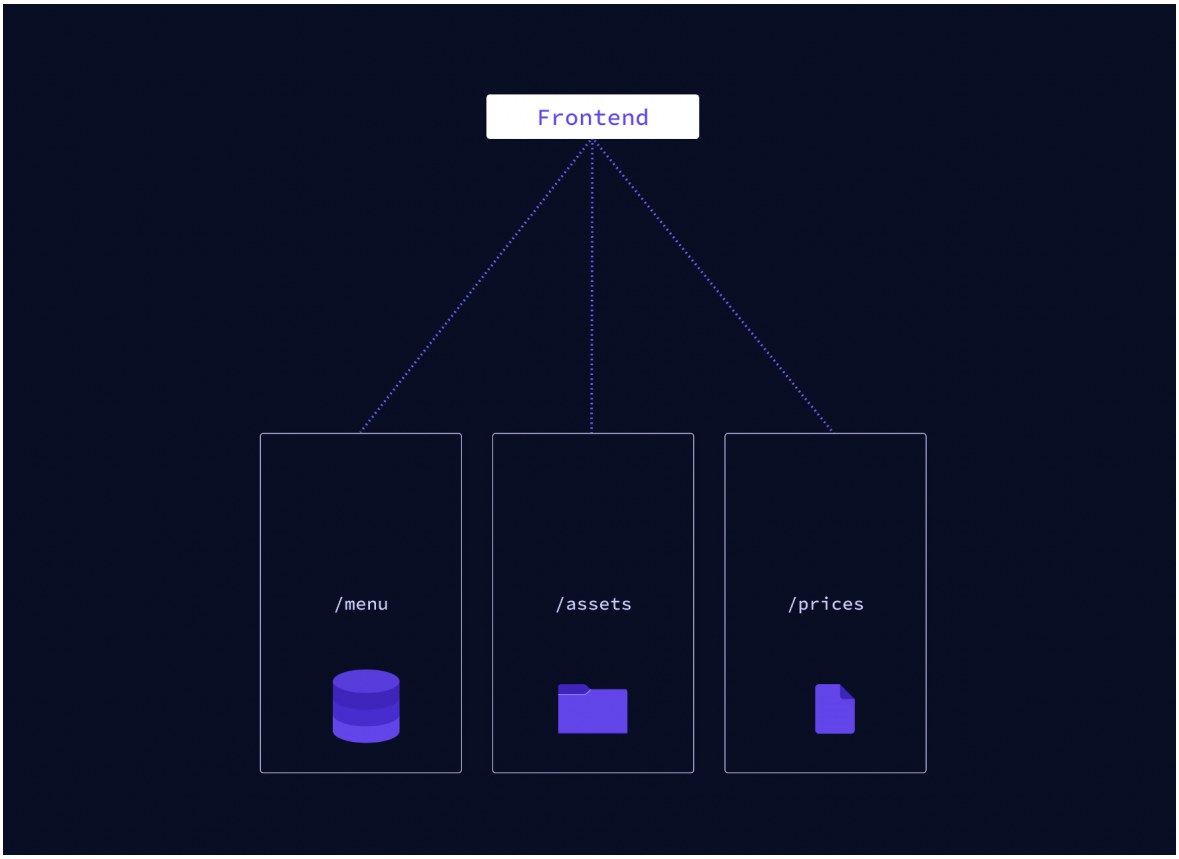
(Masse, 2011)

Appendix 3. Coupled vs Tightly Coupled



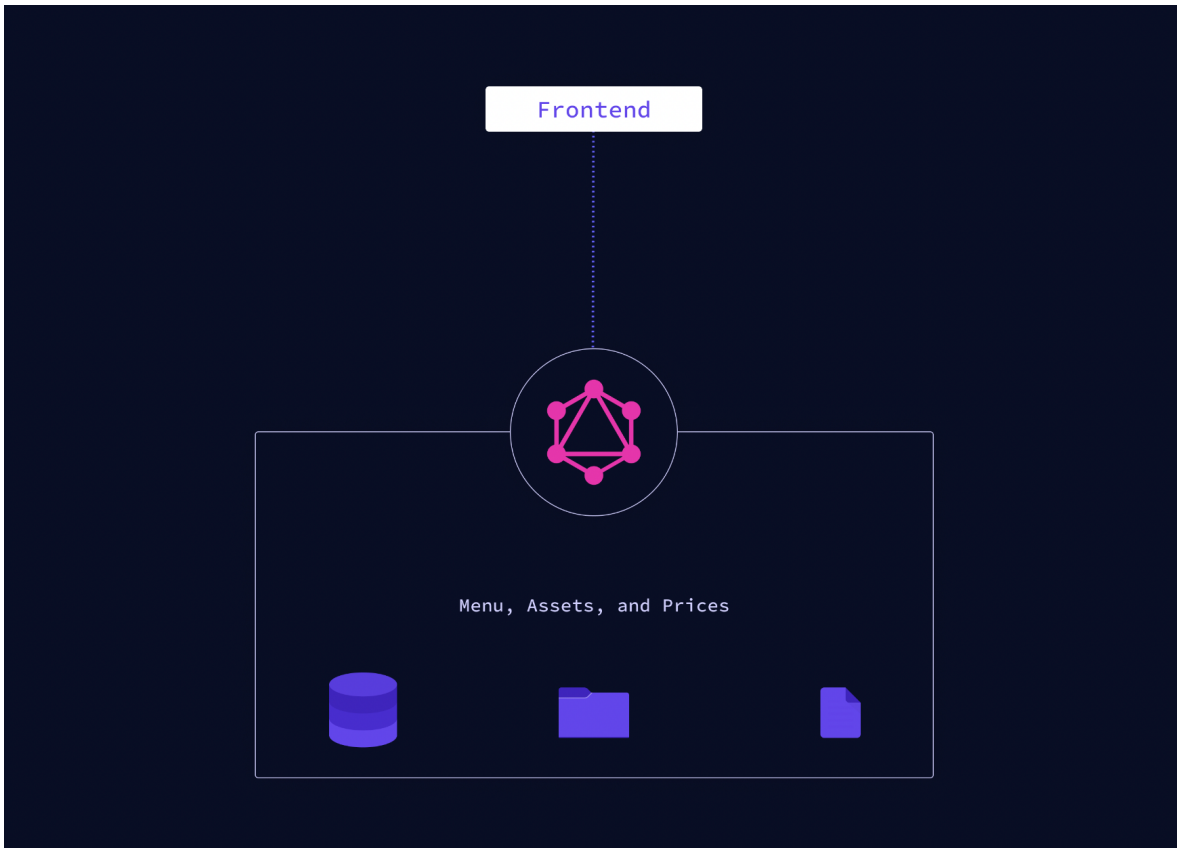
(Gilmore, 2019)

Appendix 4. The way REST APIs works with data.



R. Ganatra (2021).

Appendix 5. The way GraphQL APIs works with data

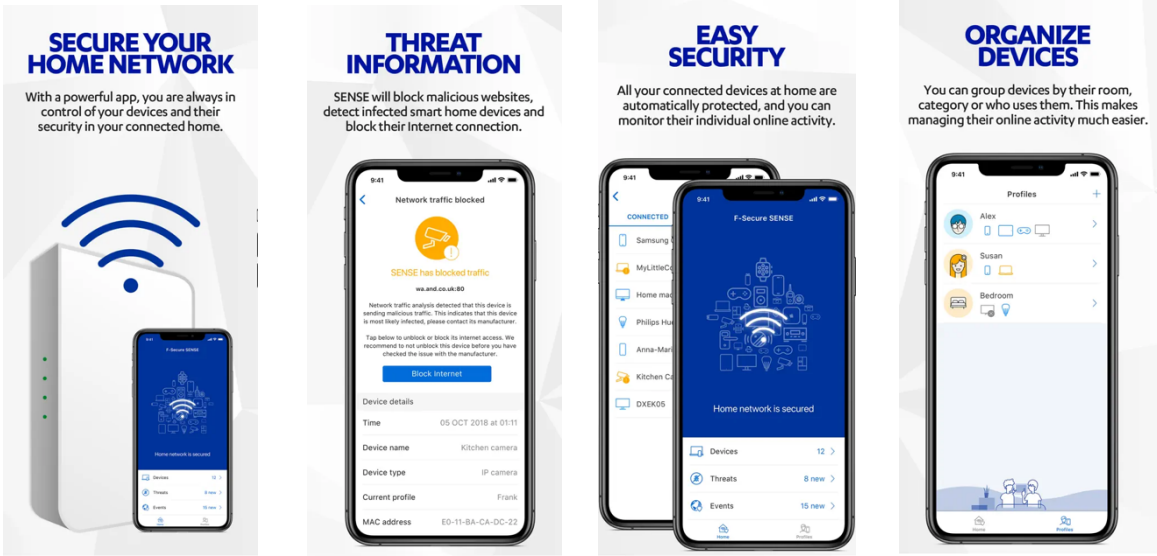


R. Ganatra (2021)

Appendix 6: Example of graphql query in SENSE app

```
query Devices($routerId: String!){
    getRouterDevices(routerId: $routerId) {
        devices {
            smartHomeDevice
            detectedManufacturer
            detectedModel
            detectedName
            detectedOs
            detectedOsVersion
            detectedType
            deviceId
            displayName
            firstSeen
            ipAddresses
            lastSeen
            mac
            profileUuid
            eppDetected
            enforcement
        }}}
```

Appendix 7: SENSE iOS application



App Store (2023)