EXPLORING GRAPHQL AS AN ALTERNATIVE TO REST

The agnostic query language for API

Bachelor's thesis

Degree Programme in Computer Applications

Häme University of Applied Sciences

Autumn 2023

John Adekunle Josiah

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Computer Applications
Author John Adekunle Josiah
Subject Exploring GraphQL as an Alternative to REST
Supervisors Deepak KC

Abstract
Year 2022

In today's fast-paced software development world, building efficient and flexible APIs is crucial. REST had been the go-to for many developers, but GraphQL had recently emerged as a promising alternative. This thesis project aimed to explore the advantages and disadvantages of using GraphQL as an alternative to REST, helping readers make informed decisions about which approach to take for their API projects.

The thesis discussed the use cases where GraphQL was being adopted by big companies, tapping into its flexibility, performance, and speed. Additionally, in other use cases, GraphQL had been used on top of REST.

Next, the thesis started by giving an overview of REST and GraphQL, including their basic concepts and how they differed in terms of API development. It then delved into the pros and cons of GraphQL compared to REST. For instance, GraphQL allowed for more efficient data retrieval, reduced over-fetching, and offered a more flexible data model. However, there were also challenges, such as a steeper learning curve and a need for specialized tooling.

The thesis project concluded with recommendations for when to use GraphQL instead of REST, based on project-specific requirements. It also suggested tooling and ecosystem when implementing GraphQL.

In summary, this thesis project provided a comprehensive exploration of GraphQL as an alternative to REST. By understanding the benefits and drawbacks of each approach and the scenarios where they were most suitable, readers could make informed decisions about which approach to use for their API development projects.

Keywords GraphQL, REST, API development, Client-server communication, Endpoint, Over fetching, Under fetching, Query Language, Web Services.

Pages 43 pages

Glossary

HTML	Hypertext Markup Language
REST	Representational State Transfer
API	Application Programming Interface
GRPC	Google Remote Procedure Call
SOAP	Simple Object Access Protocol
XML-RPC	XML Remote Procedure Call
CORBA	Common Object Request Broker Architecture
WSDL	Web Services Description Language
XML	Extensible Markup Language
SQL	Structured Query Language
SDL	Specification and Description Language
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
JWT	JSON Web Token
OAuth2	Open Authorization 2.0
GraphQL	Graph Query Language
CRUD	Create, Read, Update, Delete
URQL	Universal React Query Library
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

CURL	Client URL
IDE	Integrated Development Environment
FQL	Facebook Query Language

Table of Contents

List of Figures

List of Codes Snippets

1 Introduction

Application programming interface (API) is used every day online and utilized by businesses and digital companies, the likes of Amazon, Google, and many more. Each time an individual makes a payment online or uses apple music to listen to music, an API channel is indirectly consumed, without the user knowing. During my internship, HAMK Smart exposed me to GraphQL and a mobile programming course at HAMK allowed me to learn about the restful API. Then I determined that my thesis subject would be exploratory on both REST and GraphQL to further my understanding of the various methods (SOAP, GRPC, XML-RPC, CORBA, WSDL) for building API and providing data to clients.

RESTful architecture was introduced in year 2000, by Roy Fielding, as a standard for the software industry, and it still dominates the field of Web APIs. Web browsers usually are the main client for Web APIs, but several types of consumers (mobile users) of web APIs continue to emerge and more variety of APIs were introduced in production use as they were scaled to extreme heights, certain problems transpired in the architecture. (readme, 2016) Facebook had issues with the REST Architecture style such as Multiple endpoints, Over-fetching, and Under-fetching of the resource. GraphQL was then designed to solve these problems. GraphQL utilizes types and fields to describe its' data and schemas; while REST requires many HTTP endpoints to fetch data which can most likely result in over-fetching, GraphQL exposes only one HTTP endpoint due to its flexibility of data structure. (Gudabayev, 2021)

The goal is to illustrate the uses and flexibility of GraphQL and investigate the differences between REST and GraphQL. The thesis activities will illustrate how GraphQL works and how GraphQL servers communicate with the Client-side. To achieve the thesis goal, the following research questions shall be answered throughout the thesis:

- What is the reason behind the development and usage of GraphQL?
- Is GraphQL truly an alternative to REST?
- How does GraphQL work, and what problem is it solving?
- How do Authentication and Authorization work in GraphQL?

2 GraphQL

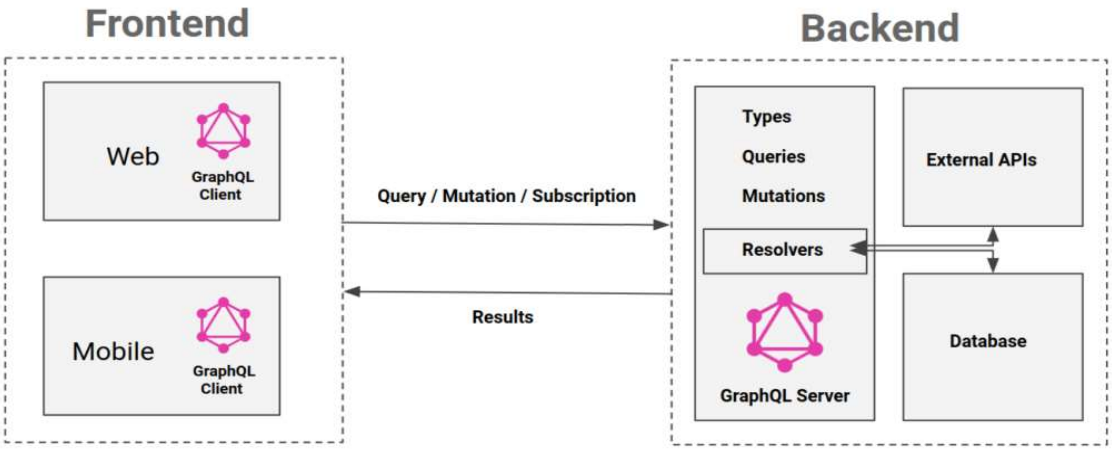
GraphQL Foundation has open-sourced the GraphQL language, which was initially developed by Facebook. This indicates that it is now covered under the MIT license, thus you may use it for commercial purposes. GraphQL is a query language for graphs, as its name implies and the second component which is QL, or the query language enable us to do fun stuff, such as CRUD (Escott, 2020). The Structured Query Language is a query language we are acquainted with (SQL). The Structured Query Language is a domain-specific language used to manage database data. GraphQL is also a domain-specific language for managing data via an API (or web services). (Escott, 2020)

GraphQL serves as both a query language and a runtime for efficiently handling requests by utilizing existing data. It furnishes a comprehensive and easily comprehensible representation of the available data in the API. It empowers clients to request precisely what they require and nothing beyond, simplifying the evolution of APIs over time. Additionally, it facilitates the use of robust developer tools (The GraphQL Foundation, n.d.).

(Clark, 2019) Clearly explains that "--- A GraphQL query represents a string that, upon being transmitted to a server, yields JSON in return to the client. These queries exhibit a mirroring effect with their responses, facilitating a straightforward prediction of the data structure resulting from the query execution. Furthermore, this characteristic simplifies the task for developers, allowing them to easily craft queries when they possess a clear understanding of the specific data requirements for their application." In Figure 1, a visual representation of The GraphQL Architecture which shows the communication flow between the frontend and backend, explains how GraphQL architecture is like a blueprint that helps developers request and receive exactly the data they need from a server.

A GraphQL service is established by defining types, fields, and functions for each field for each type. (The GraphQL Foundation, n.d.). Clients can declare exactly the data needed from an API, which should display to the user, the requests are sent to the server, and GraphQL server will only expose a single endpoint and responds to the request by retrieving the appropriate information from the database and sends it back to the client.

Figure 1 The GraphQL Architecture (Quique, 2020)



The client can send HTTP requests that includes schema like Data Types, Queries, Mutations and Subscriptions. When the request is received the server will process those requests with the help of a resolver. (Quique, 2020)

2.1 History of GraphQL

GraphQL was developed inside Facebook in 2012, when Facebook decided to rebuild the application's native mobile apps. GraphQL specification was open-sourced and made available to the public in 2015, with specification and a reference implementation called graphql.js. (The GraphQL Foundation, n.d.)

The company's iOS and Android applications were merely thin wrappers around mobile website views. Facebook has a RESTful server with FQL data tables (Facebook's equivalent of SQL). The applications struggled with performance and frequently crashed. At that moment, developers understood they needed to enhance how data was transmitted to their client applications.

The team of Lee Byron, Nick Schrock, and Dan Schafer decided to rethink their data from the client side. (Porcello & Banks, 2018). Facebook strategy (i.e., adopting HTML5 on mobile) for the News Feed implementation on mobile was not working due to the issues occurring from high network usage. The existing APIs were not designed to allow developers to expose a rich news feed-like experience on mobile. (Clark, 2019)

As a solution to the inefficient mobile experience. Facebook introduced a new News Feed API specifically for their mobile app, marking the inception of GraphQL. This technology was purposefully designed to optimize network usage by leveraging efficient data fetching capabilities. As of now, GraphQL usage has extended, and it is used for most of the Facebook IOS apps. Starting in September 2016, GraphQL was officially production-ready, many companies like GitHub, Twitter, PayPal, Yelp, The New York Times, IBM, Intuit, Pinterest, Airbnb, Coursera, Shopify, Instagram, Stack Share Tokopedia, KAVAK and more have adopted GraphQL for API purposes. The GraphQL project was then moved to the non-profit GraphQL Foundation in 2018. (Clark, 2019)

2.2 The development of GraphQL

Facebook relied on Apple and Google to compete to develop the best web-mobile browser experience. Instead, Apple focused on investing resources on the native mobile apps, while Google bothered not about proper web-mobile experience. The consumers and market steadily moving from computers to mobile usage and the prevailing operating systems available on mobile were incapable of providing the befitting user experience. In the quest to improve the user experience of Facebook on mobile, several new senior iOS engineers were hired to join forces with the in-house engineers to rewrite iOS app for Facebook; the news feed to start with. (Gudabayev, 2021)

The team of engineers realized the existing news feed API usually returns data as HTML. To follow the web and mobile standards as of 2012, It was necessary to convert the news feed API into a RESTful API. Having translated the Facebook newsfeed, the developers were faced with significant issues requiring alternate solutions. The answer to the mobile market issues encountered by Facebook and several other firms. The team's solution avoided retrieving too little or too much data and prevented the creation of an absurdly substantial number of endpoints. The solution was the birth of GraphQL. (Gudabayev, 2021)

Amongst the problems faced were, multiple requests are made to the server, it makes the load time slower and increases the frustration level of a user, which user tends to close the Facebook app. This is because the API cannot fetch all the data requested once, The iOS app

client initiates multiple requests, engaging in back-and-forth communication with various platforms and APIs.

When changes are made to public-facing API or API response format, the risk of breaking applications occurs, due to the reason that data are expected to be returned in a certain shape and there are returned differently. The workaround involves creating multiple versions of API and this creates more complexity for maintaining the API. To ensure that the accurate data is loaded precisely before rendering, engineers at Facebook did have to do a manual job maintaining the client model classes, network logic and related activities. (Mychal, 2020)

REST cannot fetch only the data wanted without the developer having to create another custom route. Usually, the entire response object gets fetched, and the developers only utilize a portion of the returned data. This scenario occurs when the routes don't provide all the necessary data needed by the client for rendering, so an extra request to the server is necessary. The changes made to the view by the client are mandatory to be handled on the server side. This can be cumbersome as the list of routes increases and it also reduces productivity and increases tasks for developers. (Masand, 2020)

2.3 Utilization of GraphQL by Digital Corporations Such as Netflix and Coursera

NETFLIX uniquely leverage GraphQL microservices to create a fully formed backend. The in-house team of developers in Netflix uses a custom Docker image running on top of a GRAPHILE toolset. GRAPHILE is a collection of tools centered on PostgreSQL and/or GraphQL that enables the rapid development of robust, performant, and extendable applications.

GraphQL was viewed as an upgraded API definition for CRUD software development. Netflix engineers could expose PostgreSQL dataset using a containerized model. The adoption of GraphQL has been a success, Netflix maintains over six hundred production services and deploys one hundred times daily. (Charboneau, 2022)

COURSERA used JSON early on for its APIs, later changed to Scala, experimented with specific "experience-based APIs," and finally developed its own Naptime REST framework. F However, both Naptime and REST presented significant challenges and difficulties. Flattened

data lists were challenging for clients to manipulate. Maintaining the Coursera URL structure over an extended period became challenging for developers. (Charboneau, 2022)

To tackle the problem, the team devised a resolution by creating a GraphQL assembler service, aimed at optimizing communication among different services. This accomplishment was realized through the integration of the language with its microservices infrastructure. The outcome is a cohesive schema that seamlessly interacts with Coursera's numerous services. This enhancement has significantly expedited data retrieval across the entire platform. (Charboneau, 2022)

2.4 Comparative Illustration of Requests and Endpoints in REST and GraphQL

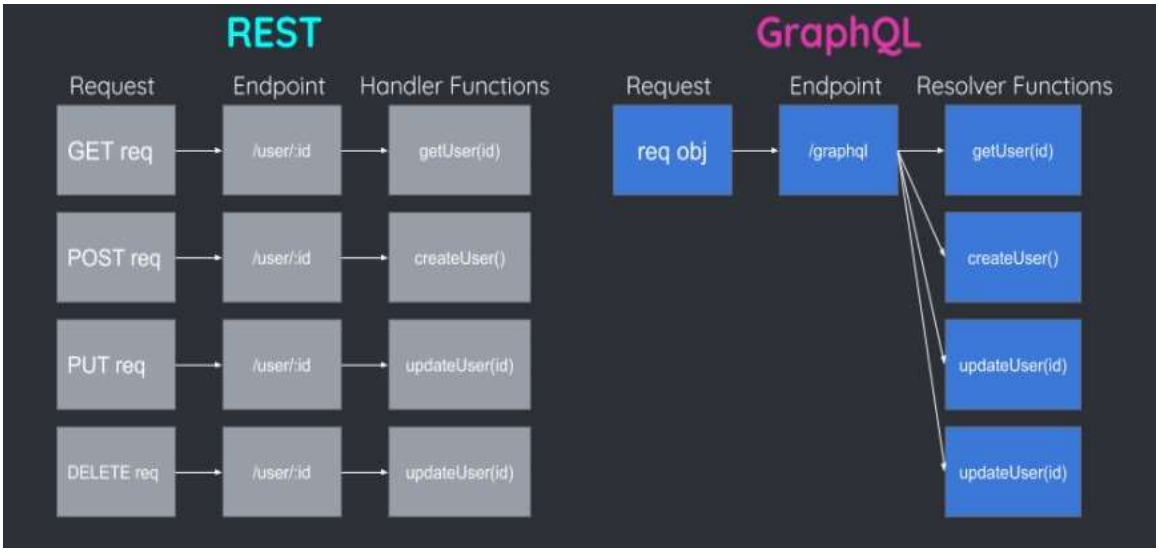
The comparative analysis of requests and endpoints in REST and GraphQL underscores the distinctions between these prevalent web API technologies. In REST, requests target specific endpoints, each corresponding to a unique resource or functionality. Server-defined endpoints necessitate clients to make multiple requests to fetch related data. Conversely, GraphQL streamlines this process by utilizing a single endpoint and a versatile query language. This allows clients to precisely request the required data, mitigating over-fetching or under-fetching issues. This comparison emphasizes the request structure and endpoint architecture in REST and GraphQL, highlighting how GraphQL's unified endpoint and query language offer enhanced flexibility and efficiency in data retrieval compared to the conventional REST approach.

REST API and GraphQL are both widely employed for web API development and consumption, each having its merits and drawbacks suitable for various applications. Rest, being the traditional method, involves multiple endpoints for data retrieval and manipulation. While effective for simple operations, it can become intricate with more complex tasks. In contrast, GraphQL's streamlined approach with a single endpoint and a tailored query language makes it particularly advantageous for scenarios requiring precise data retrieval, distinguishing it from the more conventional REST methodology.

In Figure 2, A visual representation of the fundamental architectural differences between REST and GraphQL is shown in Figure 2. This figure plays a crucial role in this thesis, acting as

a crucial point of comparison for the analysis that follows. This graphical representation will remain crucial in illustrating the basic differences and shared characteristics between REST and GraphQL as the thesis proceeds to the remaining chapters.

Figure 2 Comparative Illustration of Requests and Endpoints: REST vs. GraphQL (Mychal, 2020)



GraphQL typically serves over HTTP through one endpoint, in contrast to REST APIs that expose multiple URLs which expose single resources. (Mychal, 2020)

The intricate workings of REST are covered in more detail in Figure 3. To be more specific, Figure 3 will focus on a detailed analysis of "Handling Request Scenarios in REST Architectures." The upcoming figure will include an examination of the common procedures governing the handling of various client requests within RESTful APIs. These operations include GET, POST, PUT, and DELETE, among others, but are not limited to them.

2.5 Comparing Data Retrieval Mechanisms: REST vs. GraphQL

To illustrate the contrasting characteristics between REST and GraphQL, consider a hypothetical scenario involving a blogging application. In this scenario, the application's objective is to present a user's profile screen, which includes the user's first name, all the

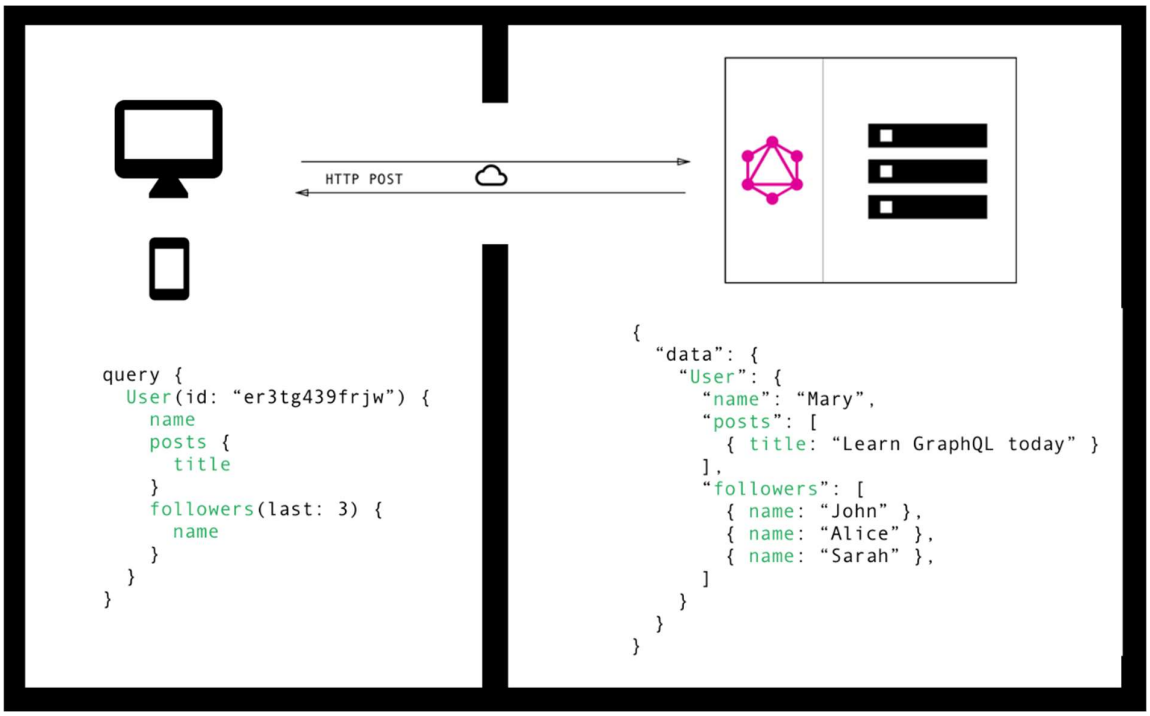
posts authored by the user, and the three most recent followers who follow the user's posts. The question arises as to how this problem can be resolved utilizing either REST or GraphQL.

Figure 3 Handling Request Scenarios in REST Architectures (Prisma & GraphQL community, 2023)

In the context of REST, the retrieval of essential data required the initiation of three distinct GET requests targeting separate endpoints. The accompanying image illustrates the corresponding endpoints, namely users/<id>, users/<id>/posts, and users/<id>/followers. These requests were made to retrieve the user's username, user posts, and followers. However, an analysis of the depicted response reveals an instance of over fetching, whereby the returned data included extraneous information that was not pertinent to the original request, such as the user's birthday and addresses.

In Figure 4, a detailed illustration of the GraphQL server's response mechanism when handling a single query is presented. This visual representation offers valuable insights into the complex process of query execution, highlighting the server's efficiency and data retrieval steps in GraphQL.

Figure 4 GraphQL server, responding to a single query (Prisma & GraphQL community, 2023)



In Figure 4, a query is shown on the left-hand side to request specifically for the objects or data that is needed and on the right is our response, which consists of JSON data prepared according to the structure of our request. This response provides only the requested information and eradicates over fetching. (Prisma & GraphQL community, 2023)

2.6 Core Concepts behind GraphQL

GraphQL is a query language for APIs developed by Facebook as an alternative to traditional REST APIs. GraphQL was released to the public in 2015 and has since become one of the most popular options for building and consuming APIs. In Figure 5, The Schema Definition Language (SDL) will be explained through the labelling of content in a GraphQL Schema. A GraphQL API is defined by a schema, which serves as the contract between the client and

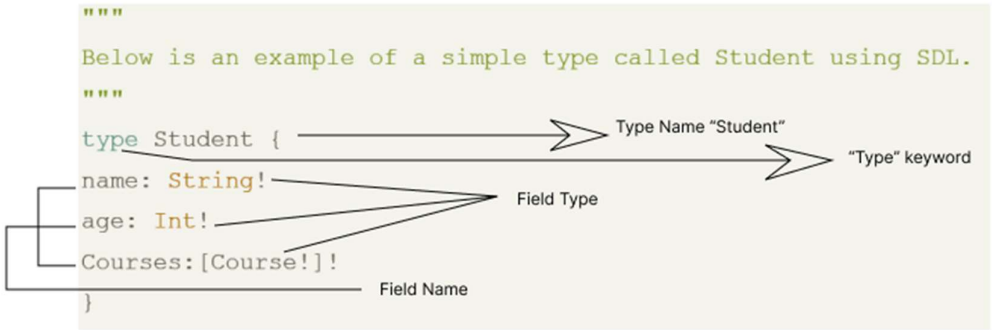
server. The schema defines the types of data that can be queried, as well as the fields and relationships between those types. In Code 1, a GraphQL schema titled "Student" is used as a demonstration of what a schema should consist of. GraphQL has its own type-system used to define an API's schema. (Prisma & GraphQL community, 2023)

Code 1. GraphQL API, Schema Definition Language(SDL).

```
"""
Example of a simple type called Student using SDL.
"""
type Student {
name: String!
age: Int!
}
```

This data type comprises two fields: "name" and "age," which are of types String and Int, respectively. The "!" following each type indicates that these fields are mandatory. (Prisma & GraphQL community, 2023).

Figure 5 The content in a GraphQL Schema.



Additionally, relationships between types can be expressed. A Student could be associated with the Courses he/she is taking in school. GraphQL has a strongly type system, which defines the structure of the data that can be queried or mutated. This allows the client to make requests for exactly the data it needs, rather than receiving a larger, more structured response that includes unnecessary data. In Code 2 Object Types specify the fields (properties or attributes) that can be queried on that type and the data types of those fields.

Types can be scalar types (e.g., String, Int, Boolean), enumeration type, interface type, union type, or custom object types (e.g., User, Post).

Code 2 Object Types in GraphQL

```
type Course {
  title: String!
  studentinfo: Student!
}
```

Similarly, the other end of the relationship needs to be placed on the student type. In Code 3 a relationship between the student and the course is depicted as a one-to-many relationship in data modeling.

Code 3 GraphQL Schema Type Student

```
type Student {
  name: String!
  age: Int!
  courses: [Course!]!
}
```

The courses field on Student is an array of posts, this is a construction of a one-to-many relationship between Student and Course. Code 4 Illustrates the process of fetching data with queries, while the responses to these queries are displayed in Code 5. Queries are used to request specific data from the API. In GraphQL, the client specifies precisely what data it need, and the server only delivers that data. This reduces the amount of over-fetching or under-fetching of data that can occur in a traditional REST API. Here is an example of a request that a client could send to a server requesting only the field “name”.

Code 4 Sample request a client could send to a server to request data

```
{
  allStudents                                                    {
    name
```

```
  }
}
```

The allStudents field in the query is called the root field, while everything that comes after the root field is a payload of the query. In Code 5, the "payload" for "allStudents" might include information about all the students, such as names, IDs, grades, or any other data associated with students, depending on how the GraphQL schema is structured. However, in this schema, only the name is requested in the payload, and that's the information returned in the response.

Code 5 Sample response a server could return to a client

```
  "allStudents": [
    { "name": "Johnny" },
    { "name": "Mathew" },
    { "name": "Enitan" }
  ]
```

Fields represent the properties of an object type in the schema. Queries can request specific fields on an object type, and the server will return only those fields in its response. (ApolloGraph, 2023). Resolvers are functions that are responsible for returning the data for a specific field in a query. They act as an intermediary between the client and the data source, retrieving the necessary data and returning it to the client. (ApolloGraph, 2023)

In Code 6, the resolvers play a crucial role in populating the response with the requested data, making GraphQL a flexible and efficient way to retrieve information from various data sources. Each field in a GraphQL schema has an associated resolver that determines how to retrieve the data when that field is queried. The resolver for retrieving students in this schema is a function that specifies how to fetch information about students when the "students" field is queried in a GraphQL request.

Code 6 Example of a resolver in GraphQL Schema

```
const resolvers = {
  Query: {
```

```
    students: () => students,
  },
};
```

In GraphQL, variables can be passed along with a query to dynamically update the data that is returned from the server. This allows for greater flexibility and reduces the amount of duplicated code in the client. (ApolloGraph, 2023)

In GraphQL, Mutations allow for creating, updating, and deleting data on the server. They are used to modify the data in the API and return a response to the client. In Code 7, the `createStudent` mutation is employed for the purpose of creating a new student. It requires an input object that includes fields like name and age. The values for these fields are provided to generate the student with the desired information. Following the execution of this mutation, a typical response should be received, which includes the newly created student's information, including their name and age. While mutations and queries have the same syntactical form, the mutation keyword must always come first. (ApolloGraph, 2023)

Code 7 A sample GraphQL mutation to create a new student

```
type mutation {
  createStudent(name: "Bob", age: 32) {
    name
    age
  }
}
```

Real-time updates with subscriptions are a feature of GraphQL that allows clients to receive notifications about changes to back-end data in real-time. Real-time subscriptions are useful for applications that require live updates, such as stockbroker applications. The same syntax, such as queries and mutations, is used when writing subscriptions. (Prisma & GraphQL community, 2023). In Code 8, the client subscribes to the newStudent subscription for a specific student with the studentID of "ABC." The client specifies which fields of the Student type it wants to receive updates for. Whenever an event happens that impacts the specified student (e.g., an update to their information), the server will push the updated data to the

client, allowing real-time updates to be displayed in the application. Subscriptions are a powerful feature in GraphQL that enable you to build real-time and event-driven applications, making it easier to keep clients updated with the latest information. The implementation details of subscriptions may vary depending on the GraphQL server and client libraries been used.

Code 8 Sample subscriptions on events happening on the Student type

```
type subscription {
  newStudent(studentID: "ABC") {
    id
    name
    age
  }
}
```

By leveraging these fundamental principles, GraphQL offers an effective system for clients to precisely request the data they need, reducing unnecessary data retrieval. It also provides a straightforward way to define the structure of data in an API, facilitating smoother interactions between clients and servers.

2.6.1 Principle of GraphQL

Perhaps because GraphQL does not dictate how we should create API; it does provide guidelines for how to conceptualize a service. GraphQL queries follow a hierarchical structure, Fields are nested within one another, and the query resembles the data it returns. (Misquissene, 2020). GraphQL requests are often more efficient than REST API calls because clients can specify precisely which data they need. The language and runtime supporting the client also contribute to the efficiency of GraphQL requests (Misquissene, 2020). GraphQL schemas are strongly typed. Each data point is linked to a predefined type, guaranteeing validation, and providing clients with accurate details regarding the available data and how it can be queried. (Misquissene, 2020).

GraphQL server enable clients to specify exactly what data are needed and eliminating the need for multiple roundtrips to the server (Misquissene, 2020) GraphQL schemas are introspective, meaning they can be queried to discover what types and fields are available. (Misquissene, 2020)

2.6.2 Key features of GraphQL

GraphQL is statically typed, which means that there is no need to create a variable prior to using it. This feature promotes efficiency and ease of use when working with GraphQL. (Hartman, 2023). It allows for a clear separation between the frontend and backend, promoting modularity and flexibility in the development process.
GraphQL avoids over- or under-fetching of data, which enhances the efficiency of queries and reduces the amount of data transferred between the client and server (Hartman, 2023).
GraphQL is language- and HTTP-agnostic, meaning it can be used with any programming language and can be accessed using various protocols. This feature promotes versatility and interoperability in the development process. (Hartman, 2023). Unlike other technologies, GraphQL documentation does not incur any additional costs. This enhances the accessibility and affordability of GraphQL documentation for developers (Hartman, 2023).

GraphQL aids in bandwidth conservation as it permits clients to request only essential data, reducing the data transmitted over the network. This, consequently, leads to decreased latency and improved overall system performance.

2.6.3 Application of GraphQL

GraphQL finds practical application in various scenarios, offering client frameworks like Relay for developers. One of its key advantages is its ability to optimize the performance of mobile applications. This is achieved by allowing the client application to precisely specify the required fields in extensive queries, thereby minimizing data transfer overhead and enhancing overall efficiency. In the domain of object-oriented programming, GraphQL commonly employs the mix-and-match facade pattern. Developers benefit from this pattern by efficiently constructing requests that retrieve data from multiple sources in a single API

call. This not only saves valuable time but also conserves bandwidth by reducing the number of necessary requests. (Hartman, 2023)

A significant use case for GraphQL is its role in simplifying complex APIs. Developers can streamline their queries by specifying only the essential fields, resulting in more efficient communication with the server. This feature contributes to improved performance and a more manageable API structure. Another notable application is the incremental implementation of functionality within existing APIs. GraphQL provides developers with the flexibility to introduce new features gradually, avoiding disruptions to the existing API structure. This incremental approach enhances maintainability and reduces the risk of introducing bugs during updates. GraphQL follows a robust approach to error handling, resolving as much data as possible before returning. In the case of errors, it returns a top-level error object with an array of errors while still providing as much data as possible. (mobilelive, 2023)

This strategy ensures that applications can gracefully handle issues, enhancing user experience by delivering maximum available information. For deployment and testing, GraphQL offers an integrated development environment (IDE) known as GraphiQL. This tool simplifies the deployment of GraphQL and assists developers in composing queries, visualizing responses, and debugging applications. Additionally, GraphQL serves as an abstraction layer on top of existing APIs, enabling developers to define the structure of the response based on specific user requirements. This flexibility allows for tailored responses without altering the underlying API, making it adaptable to diverse client needs. GraphQL's applications extend from optimizing mobile app performance to simplifying API complexities, making it a versatile and valuable tool in various development scenarios. (Hartman, 2023)

2.6.4 **Advantages of GraphQL**

GraphQL offers a versatile approach to enhance the efficiency and effectiveness of web development, providing numerous advantages that significantly impact data retrieval, development workflows, and overall, API evolution. This modern query language and

runtime system contribute to improved performance and streamlined development processes in several ways.

At the core of GraphQL's strength is its ability to empower clients to precisely specify the data they need, minimizing unnecessary data transfer over the network. By allowing clients to define the response structure, GraphQL optimizes data fetching, reducing latency and bandwidth usage. This capability is particularly beneficial for applications in resource-constrained environments, such as mobile devices with limited bandwidth and processing power. For instance, GraphQL enables a mobile application to request only essential user details, resulting in a leaner response payload and faster load times. (Bhardwaj, 2023)

The built-in schema of GraphQL facilitates dynamic and interactive exploration of available data and relationships between objects. Developers can easily navigate and understand the data model, fostering a more intuitive API development process. This feature proves invaluable during initial development stages and contributes to ongoing maintenance by providing a clear blueprint of the API structure. For example, developers can seamlessly explore and comprehend relationships between user profiles, posts, and comments within the GraphQL schema, promoting efficient API development and maintenance. (Bezugla, GraphQL: Core Features, Architecture, Pros, and Cons, 2019)

GraphQL APIs are inherently strongly typed, meaning that the data types returned by the API are explicitly defined. This characteristic enables early detection of errors during the development phase, as any deviations from the defined types trigger immediate alerts. This proactive approach to error identification contributes to more robust codebases and accelerates the debugging process. For instance, a developer attempting to access a field not part of the specified data types will receive an error, allowing for rapid identification and resolution of potential issues. (ApolloGraph, 2023)

One standout feature of GraphQL is its capacity to accommodate changes to the API schema without disrupting existing clients. This forward-compatible design allows for the seamless evolution of APIs over time. Developers can introduce new fields, modify data structures, or deprecate outdated features without causing downtime or breaking compatibility with existing applications. Consider an example where an e-commerce platform effortlessly

introduces new product attributes without affecting the functionality of a mobile application built on a previous version of the API. (ApolloGraph, 2023)

In essence, GraphQL emerges as a powerful toolset that not only optimizes data retrieval and API development but also fosters a resilient and adaptable ecosystem. By combining selective data retrieval, dynamic exploration, strong typing, and forward compatibility, GraphQL proves instrumental in creating robust, high-performance applications with a focus on long-term maintainability and evolution.

2.6.5 Disadvantages of GraphQL

While GraphQL is a good option compared to REST, it hasn't entirely taken over yet. It's worth mentioning that sending queries for specific data can also be achieved in REST using various JSON API methods. JSON schemas can be useful for employing a schema and strong types with REST. On the other hand, developing these libraries can be difficult, therefore it may be preferable to use GraphQL, which natively supports all these functionalities. Complex queries cause performance difficulties. With GraphQL, clients can ask for exactly what they need. However, if a client asks for too many nested things all at once, it might slow things down. In such cases, it's better to use a REST API, and employing multiple endpoints and specific, finely tuned queries, even though this might take more time due to multiple calls, it's safer for server upkeep. (Bhardwaj, 2023)

In addition, using rate-limiting measures to stop wasteful requests from the other side, such as maximum query depths, query complexity weighting, preventing recursion, or persistent queries, is a solution for staying with GraphQL. While GraphQL is the optimal choice for numerous microservices, it is Insufficient for tiny applications, while REST architecture is preferable for basic applications. REST can be useful for connecting resource-driven applications that do not require the query flexibility provided by GraphQL. (Javatpoint, 2022)

Caching in GraphQL web applications can present both challenges and opportunities. GraphQL on its own does not prescribe a specific caching strategy, leaving it up to the implementers or developers. But GraphQL provides specific caching libraries and tools that can aid in managing caching complexities which are Apollo Server and Relay offer caching

solutions tailored for GraphQL. Apollo or Relay clients with built-in caching techniques can be used to implement GraphQL caching at the database or client levels. GraphQL diverges from REST by not relying on the HTTP caching mechanisms that store request content. Unlike REST, where caching efficiently stores frequently accessed data near the client to minimize server traffic, GraphQL lacks a similar mechanism. (Javatpoint, 2022)

In the REST paradigm, characterized by multiple endpoints, configuring web caching based on specific URL patterns, HTTP methods, or resources is straightforward. However, GraphQL, with its single endpoint featuring various queries, faces challenges in implementing this caching model. One potential resolution involves the use of persistent GraphQL queries, facilitating the creation of a file that maps queries and IDs. This partial solution allows the client to transmit only the query's identifier and arguments to the server, which then conducts a lookup. (Gupta, 2022)

Alternatively, addressing caching concerns can involve the adoption of a batching strategy. This approach amalgamates multiple data requests from the backend and submits them in a single request, often facilitated by tools like Facebook's Data Loader. (Bezugla, GraphQL: Core Features, Architecture, Pros, and Cons, 2019).

GraphQL can introduce some additional overhead in terms of both data size and server processing. This is because clients can request very specific data, which may require additional logic on the server to fulfill those requests. To further explain the additional layer, the key components of the GraphQL ecosystem within an application will be visualized in Figure 6, and they include a GraphQL client, a GraphQL gateway, a GraphQL server, and a database-to-GraphQL server." Since GraphQL does not recognize files, its specification does not contain a file uploading function. You will not have to bother with this restriction while using REST, as you may POST or PUT any material you choose.

2.6.6 GraphQL Tools and Ecosystem

The GraphQL ecosystem is rapidly growing, offering multiple programming language support and libraries support for GraphQL. The term "GraphQL ecosystem" refers to the wide range of tools, libraries, services, and resources related to GraphQL technology. In this ecosystem, GraphQL servers act as the backend, helping to expose GraphQL APIs to client applications.

Popular server frameworks, like Apollo Server and Express GraphQL, play a big role here. (Clark, 2019)

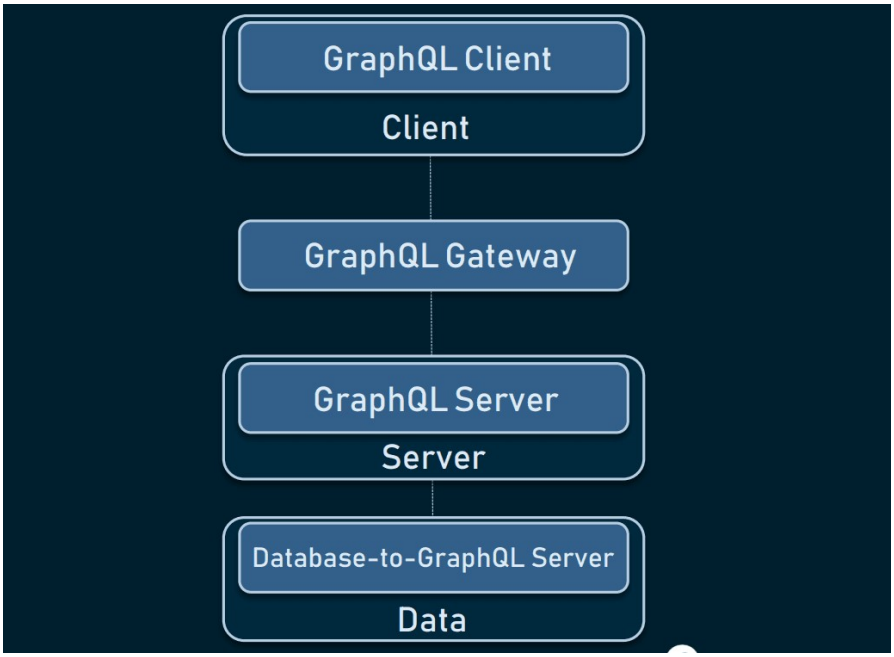
Client libraries, such as Apollo Client and Relay, help applications communicate with GraphQL APIs. The development toolbox includes GraphQL Integrated Development Environments (IDEs), code generators, and debugging tools, which make it easier to work with GraphQL. One of the strengths of GraphQL is its ability to gather data from various sources like databases, REST APIs, and third-party services. This requires data connectors and resolvers. Middleware components and plugins also come into play, allowing developers to add features like authentication, caching, and logging to GraphQL servers. (The GraphQL Foundation, n.d.)

The GraphQL community provides forums, comprehensive documentation, and a support network for knowledge sharing and problem-solving. In addition, there are hosting and deployment services that simplify the process of getting GraphQL servers up and running. Data management and caching solutions help manage and optimize data on the client side. To ensure the reliability and performance of GraphQL APIs, there are testing and monitoring tools within the ecosystem. (ApolloGraph, 2023)

The flexibility of GraphQL is highlighted through various extensions and integrations tailored to specific needs, such as real-time updates and tools for visualizing data. The GraphQL ecosystem provides a plethora of resources aimed at facilitating the development, upkeep, and optimal utilization of GraphQL APIs. This variety of tools serves to underscore the significance of GraphQL as a potent technology for constructing data-driven applications. In this context, refer to Figure 6 for further details on specific tools and resources within the GraphQL ecosystem. (Prisma & GraphQL community, 2023)

There are open-source communities building and improving the tools available for using GraphQL. In a basic application, it consists of a client, a server, and a data layer. GraphQL presents itself four times as a GraphQL client, a GraphQL gateway, a GraphQL server, and a database-to-GraphQL server.

Figure 6 GraphQL ecosystem breakdown within an application: key components (Bezugla, GraphQL: Core Features, Architecture, Pros, and Cons, 2019)



As the GraphQL community continues to expand, new tools and ecosystems are being created to make working with GraphQL even easier.

Here are some of the most used tools and ecosystems in the GraphQL community:

- Apollo Server: A production ready GraphQL server that can be used with Node.js.
- GraphQL Playground: A graphical user interface (GUI) that allows developers to explore and test GraphQL APIs.
- Prisma: A GraphQL ORM that generates GraphQL schemas and provides a type-safe database access layer.
- GraphQL Code Generator: A tool that generates TypeScript or Flow types based on a GraphQL schema, making it easier to work with GraphQL data.
- GraphiQL: A web-based IDE that allows developers to interact with GraphQL APIs.

- Relay: A GraphQL client that was originally developed by Facebook and is now maintained by the GraphQL community.
- Hasura: An open-source engine that can turn any Postgres database into a real-time GraphQL API.

2.6.7 Guidelines for Implementing a GraphQL API

GraphQL is a query language for APIs, designed by Facebook. It offers a more efficient, robust, and customizable alternative for traditional REST APIs. In Code 9, A detailed description of how to implement a GraphQL API will be demonstrated and discussed.

Initially the schema should be defined. The sorts of data the API can return and the activities that can be conducted on that data are specified in a GraphQL schema. The schema can be defined using the GraphQL Schema Definition Language (SDL), which is a simple syntax for describing the structure of your API. (Porcello & Banks, 2018)

Secondly, the Implementation of the resolvers. A resolver is a function that retrieves the data for a specific field in the schema. Each field in your schema needs a resolver written for it. Resolvers can retrieve data from a database, call an external API, or perform any other necessary actions. (Porcello & Banks, 2018)

Following the implementation of the resolver in Code 9, the server should be configured or setup. The server will be set up to manage GraphQL requests. There are many server-side frameworks and libraries available for the different programming languages. Some popular options include Apollo Server, GraphQL Yoga, and Express GraphQL.

Defining the GraphQL query and mutation operations. GraphQL queries are used to retrieve data from an API, while mutations are used to modify data. The query and mutation operations will need to be defined so that the API will support it in the schema.
The API should be tested thoroughly to ensure that it is working as expected. Tools like GraphQL Playground or GraphiQL can send queries and mutations to the API and view the responses. (Porcello & Banks, 2018)

Finally, documenting the API so that other developers can understand how to use it. There are available tools like Swagger or GraphQL Code Generator to generate API documentation automatically. (The GraphQL Foundation, n.d.)

Code 9 How to implement a GraphQL Server using Apollo Server (ApolloGraph, 2023)

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';


const typeDefs = `#graphql
 # Comments in GraphQL strings (such as this one) start with the
hash (#) symbol.

# This "Book" type specifies the fields that may be queried for
each book in our data source.
  type Book {
    title: String
    author: String
  }

# The "Query" type is unique in that it specifies all of the
queries that clients can execute, along with their respective
return types. In the above example, the "books" query returns an
array of zero or more Books.
  type Query {
    books: [Book]
  }
`;


const books = [
    {
        title: 'The Awakening',
        author: 'Kate Chopin',
    },
    {
        title: 'City of Glass',
        author: 'Paul Auster',
    },

];

const resolvers = {
    Query: {
        books: () => books,
    },
};


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


const { url } = await startStandaloneServer(server, {
    listen: { port: 4000 },
});
```

```
console.log(`🚀  Server ready at: ${url}`);
```

2.6.8 Methods for Consuming a GraphQL API

To consume a GraphQL API, there are several ways to do it and some of the most common ways are, using GraphQL client libraries to interact with a GraphQL API. Some examples of popular GraphQL client libraries are Apollo Client, Relay, and Urql. These libraries enable sending queries and mutations to the API and handling the responses. Alternatively, Fetch and Axios are commonly used JavaScript libraries to perform HTTP requests. These libraries are also suitable for sending a POST request to the API endpoint of a GraphQL API, where the query or mutation is included in the request body. (Stemmler, 2020)

CURL is a command-line tool that enables users to make HTTP requests. It can make requests to a GraphQL API by sending a POST request containing the query or mutation to the API endpoint in the request body. (Stemmler, 2020)

GraphQL Integrated Development Environments (IDEs) like Apollo Explorer and GraphiQL are tools that developers can use to work with GraphQL APIs. These IDEs allow developers to send queries and mutations to the API and handle the responses. They have a user-friendly graphical interface that makes it easy to explore the structure of the API and test queries and mutations. You can access these IDEs from a web browser, which makes them very convenient to use. (The GraphQL Foundation, n.d.)

3 Rest APIs

According to Gupta (2022), Representational State Transfer (REST) serves as an architectural style for distributed hypermedia systems. Over the years, REST has gained immense popularity and has proven its efficacy by empowering numerous prominent websites across the internet. REST adheres to guidelines that dictate how resources should be accessed and served to clients, ensuring consistency and standardization in the development of web services. By providing a predictable and consistent approach, REST simplifies web development processes. The prominent advantages of REST APIs include comprehensive documentation, ease of development and maintenance, as well as accessibility from any location. (Gupta, 2022)

3.1 Core Concepts behind REST APIs

The main concept behind REST APIs is to use HTTP requests to perform various CRUD (Create, Read, Update, Delete) operations on resources, where each resource is identified by a unique URL. There are four components a request comprises of, and they are the Endpoint which is the URL that contains the structure (root-endpoint/?) and the Method which are the types of available methods of a REST API (GET, POST, PUT, PATCH, DELETE). The Headers is another concept which fulfills a variety of functions, such as authentication and providing information on the body's content (-H or --header option can be used to send HTTP headers) and Lastly is the Data. The data is the body sent to the server, using the -d or --data option with POST, PUT, PATCH, or DELETE requests. (Husar, 2022)

3.1.1 Principles of REST API

In REST, client-server decoupling enables the separation of concerns between the client and server, fostering independence and portability of the components, with the client software only requiring knowledge of the URI of the requested resource, without any further interaction with the server application. This architecture effectively separates the user interface (client) from the data storage and business logic (server), allowing for scalability and potential cost savings (Hati, 2023). Additionally, statelessness is a fundamental principle in REST, where the API refrains from storing client context on the server between requests.

This simplifies server components, reduces network data transmission, and enhances scalability (Hati, 2023). Another important principle is a uniform interface that defines the communication between the client and server, ensuring a consistent and predictable interaction by employing standard methods such as GET, POST, PUT, DELETE, among others (Hati, 2023). Moreover, clients can cache responses from REST APIs, avoiding redundant requests, improving performance, reducing server load, and enhancing response times. REST APIs can be designed with multiple layers, allowing for easy maintenance and scalability (Hati, 2023).

In certain cases, REST APIs may employ the "Code on Demand" principle, temporarily extending or customizing client functionality by sending executable code. However, this usage is rare, and REST APIs typically provide static resources, such as Java applets, which are executed as needed (Hati, 2023).

3.1.2 Key features of REST API

REST APIs are web services that enable communication between software systems and applications. They provide a set of features that enable developers to create more efficient and dependable applications than traditional HTTP-based systems. (Hartman, 2023)
REST offers a uniform interface that defines the API structure, employing JSON for data exchange and URIs for resource identification. REST's services are highly scalable, ensuring they can readily handle increased customer demand.
Resources are easily accessible by name, and the REST API facilitates seamless data transmission, reception, and exchange within systems. It allows for the swift mapping of an application's database resources using REST API endpoints and supports the efficient storage of frequently accessed data in memory. With its simple architecture and patterns, REST offers an elegant solution for API design. Additionally, the REST API can be served from multiple servers, enhancing its versatility and robustness. (Hartman, 2023)

3.1.3 Methods of REST API

There are a few components that make up an API call. A REST API request is composed of the endpoint, the method, the headers, and the contents (or body).

The REST method which is one of the components of an API call; the Method is the HTTP-based action that a query is designed to do. It can take the form of a GET, a POST, a PUT, and DELETE.

The GET method primarily serves as a mechanism to retrieve data or information via HTTP from the server and focuses on optimizing performance. The method is safe and idempotent method. When multiple similar requests are made, It will produce the same result as a single request. (HTTP, 2023). In simpler terms, the GET method is used to retrieve information or data from a server by sending a request to a specific URL. The server then responds to the request by sending back the requested information, typically in the form of a JSON or XML representation. (Richardson & Ruby, 2007)

The POST method is used to submit an entity to a specified resource, often causing a change in state or side effects on the server. When a client submits a POST request, the server may create a new resource, update an existing resource, or perform some other side effect based on the information contained in the request. The request payload of a POST request can be in any format, but the server must have prior knowledge of how to process the payload. The server can also respond with a status code indicating the success or failure of the request, along with an optional message body that provides more details. (Gourley & Totty, 2002)

The PUT method is a common HTTP method used for updating resources on the web. A resource indicated by a URI is updated using the PUT method, which "requests that a new representation be wrapped in the request message payload."

In other words, when a client sends a PUT request to a server, it is asking the server to update the resource at the specified URI with the data included in the request payload. This can be useful for updating information about a user, changing the contents of a blog post, or modifying any other resource that can be represented as data. (Richardson & Ruby, 2007)

The DELETE method is a commonly used HTTP method that is used to request the removal of a resource identified by a specific URI. According to the HTTP/1.1 specification, the DELETE method is used to request that a resource be removed. Once a resource has been deleted, clients should not expect to be able to retrieve it from the server again. (Roy & Fielding). According to Roy T. Fielding, who is the inventor of the Representational State Transfer (REST) architectural style, "DELETE is used to delete a resource identified by a URI. The origin server SHOULD send a 202 (Accepted) status code if a DELETE method is successfully used

but the action has not yet been carried out, or a 204 (No Content) status code if the action has already been carried out and no further information is required.

3.1.4 Application of REST

REST is used to create web services that can be accessed by other applications over the internet. This allows for seamless integration between different systems and platforms. REST is used to create APIs that can be used by mobile applications to access data from a server. This allows for the creation of mobile applications that can interact with web services and provide a seamless user experience. REST can be used to create APIs that can be used by IoT devices to interact with other devices and systems. This allows for the creation of smart homes, smart cities, and other IoT applications. It is used to create APIs that can be used to access cloud-based services such as storage, computing, and databases. This allows for the creation of scalable and flexible cloud-based applications. REST is used to create APIs that can be used by e-commerce applications to interact with payment gateways, shipping services, and other third-party services. This allows for the creation of seamless and secure e-commerce experiences.(Gupta, 2022)

3.1.5 The Pros and Cons of REST

REST is easy to understand and implement, as it uses a simple set of HTTP methods (GET, POST, PUT, DELETE) and resource identifiers (URIs) to perform CRUD (Create, Read, Update, Delete) operations on resources. REST is highly scalable due to its statelessness, which allows it to handle large volumes of requests without affecting performance. RESTful APIs are reliable, as they use HTTP status codes to communicate the success or failure of a request and can use idempotent methods (PUT and DELETE) to ensure that requests are processed only once. Meanwhile the weaknesses of REST can be inflexible, as it requires clients to make multiple requests to fetch related resources, leading to over-fetching or under-fetching of data, which can result in inefficiency and slow performance. (Husar, 2022)

RESTful APIs can encounter versioning issues, as changes to resources or URIs can break client applications, leading to compatibility issues. RESTful APIs can have complex queries, as

they often require clients to construct complex URLs to request data, which can be difficult to manage and maintain.

3.1.6 Methods for Consuming a REST API

Consuming REST APIs is one of a fundamental skill for developers building web applications.
To consume a REST API, the first thing to do is to identify the endpoint URL for the API and the required parameters for the API call. The endpoint URL is the root URL for the API, while the parameters are additional data that may be required for the API call to function properly. Once the endpoint URL and parameter are identified, the request URL can be constructed by appending the parameters to the endpoint URL. (Richardson & Ruby, 2007)

To make the API call, a HTTP method such as GET, POST, PUT, DELETE, and others can be utilised. These methods are used to perform CRUD (Create, Read, Update, Delete) operations on resources, which are identified by unique URIs. When making a GET request, for example, The URI is specified for the resource to be retrieved, along with any required parameters, in the request URL.
In addition to specifying the request method and URI, you may also need to provide headers and a message body for the API call. Headers are metadata that provide additional information about the request, such as the content type, authorization information, and caching directives. The message body, if required, contains data that is sent with the request, such as JSON or XML data. (Richardson & Ruby, 2007)

Once the request URL is constructed, request method specified, and provided any required headers and message body, the API call can be made using a client library or tool such as Curl or Postman. The response from the API will typically include a status code indicating the success or failure of the request, along with any data or error messages.
Consuming REST APIs involves identifying the endpoint URL and required parameters, constructing the request URL, specifying the request method, providing any required headers and message body, and handling the response from the API. By following these steps and using appropriate client libraries or tools, developers can effectively consume REST APIs and integrate them into their web applications. (Richardson & Ruby, 2007)

3.1.7 Guidelines for Implementing a REST API

It will be smart to determine the resources that the API will expose. The resources are the entities that the API interacts with, such as users, orders, or products.
For each resource, define a set of endpoints that correspond to the CRUD operations (Create, Read, Update, and Delete) that can be performed on the resource. For example, the endpoints for a user's resource might include /users (GET) to retrieve a list of users, /users/:id (GET) to retrieve a specific user, /users (POST) to create a new user, /users/:id (PUT) to update an existing user, and /users/:id (DELETE) to delete a user. (Richardson & Ruby, 2007)

Writing a server-side code to implement the defined endpoints. This typically involves handling incoming HTTP requests, validating request parameters, performing the necessary CRUD operations on the corresponding resource, and returning an appropriate HTTP response to the client.
Defining the format of the responses that the API will return. This includes specifying the content type (e.g., JSON, XML) and the structure of the data returned by each endpoint. (Richardson & Ruby, 2007)
Defining the error responses that the API will return in case of errors or invalid requests. This includes specifying the HTTP status codes (e.g., 400 Bad Request, 404 Not Found) and the error messages that will be returned to the client. (Richardson & Ruby, 2007)
Documenting the API to provide guidance to developers who will be using the API. This includes documenting the resources, endpoints, request and response formats, error handling, and any authentication or authorization requirements. (Richardson & Ruby, 2007)

Lastly is to test the API to ensure that it behaves as expected and returns the correct responses for various scenarios.

When implementing a REST API, it is important to follow best practices and adhere to standards such as the Richardson Maturity Model to ensure that the API is scalable, maintainable, and interoperable. (Richardson & Ruby, 2007)

3.2 Comparative Analysis of GraphQL and REST APIs

GraphQL, on the other hand, is a more recent approach that provides a single endpoint to access data. It also allows you to specify precisely what data you require, eliminating the need to query multiple endpoints to obtain the desired data. This approach is great for more complex operations and can be optimized to send fewer requests and reduce response time. GraphQL core building blocks are the schema that serve as a layer between server and client while defining the acaccessibility of the data.

The schema defines the type of data that can be fetched, and the relationships between these types of data, and the type of queries that are allowed. (Tomar, 2022). The other two fundamental parts of GraphQL are Query and Resolver. The request to fetch a particular data is called a query and a resolver is used to tell the server from where and how to fetch the data corresponding to the given field. GraphQL is composed of three parts: a query for reading data, a mutation for writing data, and a subscription for receiving real-time data over time. (Tomar, 2022)

The key advantages of GraphQL are that it is more efficient than REST, enables complicated queries, and is more adaptable. Both REST and GraphQL have their strengths and are suitable for different use cases. The choice between them depends on the specific requirements of a project and the trade-offs that need to be made in terms of complexity, efficiency, and control over data fetching. In Table 1, key differences are outlined for both REST and GraphQL to illustrate their suitability for different use cases.

Additionally, As specifications for constructing and consuming APIs, GraphQL and REST share similarities. They can both fetch resources by submitting queries, return JSON data in the request, and operate over HTTP. Also, REST endpoints are like GraphQL fields in that they are data entry points that invoke server functions. Notable distinctions begin with the conceptual models: Unlike REST, which is composed of files, GraphQL is composed of graphs. GraphQL has changed developers experience of building and consuming an API.

Table 1 Key differences between GraphQL and REST technologies. (Tomar, 2022)

GraphQL Query Language	RESTful Architecture
GraphQL is Client-Driven architecture	REST is Server Driven architecture
GraphQL uses SCHEMA and STRICT TYPECASTING to organize queries	Rest utilizes ENDPOINT to deliver resources
Single call is used to call specific data	Multiple calls are used to call fixed data
GraphQL uses QUERY, MUTATION, and SUBSCRIPTION for data manipulation	REST uses GET, POST, PUT, and DELETE for data manipulation
GraphQL eliminates under-fetching and over-fetching	REST over-fetches and under fetches data
GraphQL is a growing community	REST is a large community
GraphQL development speed is faster	REST development speed is slower
It has a difficult learning curve	It has a moderate learning curve
Consistency is maintained across all platforms	Consistency across platforms is hard.
It uses metadata for query validation	It does not have machine-readable metadata cacheable.
Multiple microservices mobile apps	Simple apps resources-driven apps

4 Mechanisms of Operation for GraphQL Clients and Servers

In a GraphQL setup, the client communicates with the server through a single endpoint, typically a /graphql route. The client sends a request to the server containing a GraphQL query, and the server responds with a JSON object that contains the requested data. There is communication between the client and server that happens over HTTP. The client sends a POST request to the server containing the GraphQL query, and the server responds with a 200 OK status code and a JSON object that contains the requested data. The client can also send variables along with the query to dynamically update the data that's returned from the server.

The query sent from the client to the server describes the data that the client needs. The server processes the query, retrieves the necessary data, and returns a response that matches the structure of the query. This allows for greater flexibility and control over the data that's returned, compared to a traditional REST API where the client must make multiple requests to retrieve all the necessary data.

In summary, the communication between a GraphQL client and server happens through a single endpoint, and the client sends a query that describes the data it needs, and the server returns a JSON object that matches the structure of the query.

4.1 Procedures for Retrieving Data from a GraphQL API

In React, you can use the Apollo Client library to fetch data from a GraphQL API. The Apollo Client provides a simple and flexible way to interact with a GraphQL server.
GraphQL is a query language that allows you to fetch data from an API in a more efficient and flexible way than traditional REST APIs. In Code 10, will depict a query that fetches the name and email address of a user.

There are many client libraries available for different programming languages and platforms, such as Apollo Client, Relay, and URQL.
The endpoint URL for the GraphQL API that should be query must be known. This is typically provided by the API provider or documented in their API documentation.

GraphQL queries are written using a special syntax that allows us to specify the exact data we want to fetch from the API. we can use tools like GraphiQL or GraphQL Playground to test and refine our query.

Code 10 Example query that fetches the name and email address of a user:

```
query {
  user(id: "123") {
    name
    email
  }
}
```

In Code 10, The query sends the GraphQL query to the API endpoint using client library. The client library will handle things like HTTP requests and responses, and parsing the response data into a format that can be use in the application.

Apollo Client is a powerful JavaScript library used for working with GraphQL APIs in client-side applications. It provides a set of tools and utilities for managing data, making queries, and handling mutations in a GraphQL-driven application.

The JavaScript code in Code 11, uses the Apollo Client library to interact with a GraphQL server. It starts by importing necessary functions and objects from the library.

Next, it sets up an Apollo Client that connects to a specific GraphQL server URL and uses an in-memory cache to store data. The code defines a GraphQL query named `GET_USER`, which requests information about a user with the ID "123," specifically their name and email. It then sends this query to the GraphQL server using the `client. Query()` function. If the query is successful, the result is logged to the console. If an error occurs, it catches and logs the error.

Essentially, the code demonstrates how to use Apollo Client to fetch data from a GraphQL server and handle the response or errors.

Code 11 Example using Apollo Client in JavaScript. (ApolloGraph, 2023)

```
import { ApolloClient, InMemoryCache, gql } from
'@apollo/client';

const client = new ApolloClient({
  uri: 'https://example.com/graphql',
  cache: new InMemoryCache()
});

const GET_USER = gql`
  query {
    user(id: "123") {
      name
      email
    }
  }
`;

client.query({
  query: GET_USER
})
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

A JavaScript snippet sample of how to use Apollo Client would be illustrated in Code 11. Usually, there is a need to set up a GraphQL endpoint, define the queries and mutations, and configure the client to interact with the GraphQL server, that offers a range of features to simplify the process of working with GraphQL in the client-side applications and provides efficient data management and real-time capabilities.

4.2 Authentication and Authorization in GraphQL

In GraphQL, authentication and authorization are important aspects of securing your API and ensuring that only authorized users have access to certain resources.

Authentication is the process of verifying the identity of a user or client. In the context of GraphQL, this typically involves verifying that a user has provided valid credentials such as a username and password. Once a user has been authenticated, they can be authorized to access certain resources based on their permissions. (ApolloGraph, 2023)

To implement authentication and authorization in GraphQL, you can use various tools and libraries such as HTTP headers and JSON Web Tokens (JWT), OAuth2, and GraphQL Shield.

In Code 12, “the code extracts a user token from the HTTP Authorization header included in each operation request. It then fetches the corresponding user object for that token and adds it to the contextValue object that's passed to every executing resolver. Each resolver can then use this object to determine what data the user has access to.”

Code 12 Example of an HTTP Authorization request in GraphQL. (ApolloGraph, 2023)

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from
'@apollo/server/standalone';

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

const { url } = await startStandaloneServer(server, {
  // For `startStandaloneServer`, the `req` and `res` objects are
  // `http.IncomingMessage` and `http.ServerResponse` types.
  context: async ({ req, res }) => {
    // Get the user token from the headers.
     const token = req.headers.authorization || '';

    // Try to retrieve a user with the token
     const user = await getUser(token);

    // Add the user to the context
     return { user };
```

```
  },

});

console.log(`🚀 Server listening at: ${url}`);
```

The contextValue is generated separately for every new request, we don't have to worry about cleaning up user data at the end of an operation's execution. (ApolloGraph, 2023)

Authorization is the process of determining what actions a user is allowed to perform or what piece of data a user can view, based on their authenticated identity and their associated permissions. In GraphQL, authorization can be implemented using various strategies such as role-based access control (RBAC) or attribute-based access control (ABAC).

Let’s view the most basic type of approach to authorization in Code 13, which is an all-or-nothing approach. This approach may only be used in extremely restrictive environments that give no public access to the API, such as an internal tool or a microservice that should not be accessible to the public. The approach may only be used in extremely restrictive environments that give no public access to the API, such as an internal tool or a microservice that should not be accessible to the public. (ApolloGraph, 2023)

Code 13 Type of authorization that gives a GraphQLError if or when the user is not authenticated. (ApolloGraph, 2023)

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from
'@apollo/server/standalone';
import { GraphQLError } from 'graphql';

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

```
const { url } = await startStandaloneServer(server, {

  context: async ({ req }) => {
    // get the user token from the headers
    const token = req.headers.authorization || '';

    // try to retrieve a user with the token
    const user = getUser(token);

    // optionally block the user
    // we could also check user roles/permissions here
    if (!user)
      // throwing a `GraphQLError` here allows us to specify
an HTTP status code,
      // standard `Error`s will have a 500 status code by
default
      throw new GraphQLError('User is not authenticated', {
        extensions: {
          code: 'UNAUTHENTICATED',
          http: { status: 401 },
        },
      });

    // add the user to the context
    return { user };
  },

});

console.log(`🚀 Server listening at: ${url}`);
```

The only difference from the basic context function above is that it checks for the user. If no user exists or if lookup fails, the function throws an error, and the corresponding operation is not executed. (ApolloGraph, 2023)

5 Research Methodology

The research methodology involves a review of relevant literature on both GraphQL and REST to determine the strengths and weaknesses of each approach, as well as to identify possible use cases where GraphQL may be a better fit.
The literature review was intended to provide readers with understanding and insight into the thesis theme. The review was conducted using a systematic approach that involves identifying relevant articles, books, and other sources related to GraphQL and REST.

The review focus on recent literature, with an emphasis on peer-reviewed articles and other reputable sources. The literature review was organized thematically, with sections devoted to key topics such as the architecture and design principles of GraphQL and REST, the benefits and drawbacks of each approach, and potential use cases where GraphQL may be more effective, as GraphQL is more focused on in the entirety of the thesis.

The research design involves a qualitative approach, using a literature review to gather data and analyse the findings. The research was conducted using a mix of primary and secondary sources, with a focus on recent literature and peer-reviewed articles. The data analysed using a thematic approach, with key themes and patterns identified and analysed in relation to the research questions.

The literature review research will be conducted using a variety of sources, books, courses, including online databases, academic journals, websites, and other reputable sources.

The analysis will involve a detailed examination of the similarities and differences of both GraphQL and REST, as well as an exploration of potential use cases where GraphQL may be a better fit. The sources utilized in the research is well referenced, and the thesis is equipped to enable sources to be search by citations, which enables the retrieval of relevant articles.

The research provides valuable insights into the benefits and drawbacks of using GraphQL as an alternative to REST in web application development. The findings will be presented in a clear and concise manner, with a focus on identifying key themes and patterns related to the research questions. The research will contribute to the existing literature on GraphQL and REST, providing a valuable resource for developers and other stakeholders involved in web application development.

6 Results

After exploring REST and GraphQL as alternative choices for designing web APIs, it can be concluded that both technologies have their own merits and downsides. In the realm of data representation, REST delineates the accessible data space using a linear list of endpoints, whereas GraphQL portrays it through a schema that incorporates relationships. One of the limitations of REST is that complex queries often require multiple requests to retrieve all the necessary data. On the other hand, GraphQL allows for handling complex queries with a single request, thereby avoiding performance issues that can arise with REST when transferring enormous amounts of data. GraphQL ability to retrieve only the required data contributes to improved performance.

Despite their differences, REST and GraphQL share fundamental concepts. Both rely on frameworks and libraries to handle networking complexities. While REST employs endpoints, GraphQL uses fields to invoke server functions. A GraphQL API backend can be more complex to develop than a REST API backend. If the added complexity of GraphQL is unnecessary or unjustifiable, using REST would be a wise decision. GraphQL should be Considered if, for instance, the needs for the API layer cannot be met using REST.

Through this exploration, I have come to realize that the selection between REST and GraphQL depends on the specific requirements and nature of the project at hand. REST remains a reliable and widely adopted solution, especially for scenarios that prioritize simplicity and adherence to established conventions. In contrast, GraphQL shines when data flexibility, efficiency, and fine-grained control are paramount. Careful consideration of project goals, team expertise, and long-term scalability should guide the decision-making process. Overall, this journey of exploration has provided me with valuable insights into the nuanced differences between REST and GraphQL, enabling me to make informed decisions in future API development endeavors.

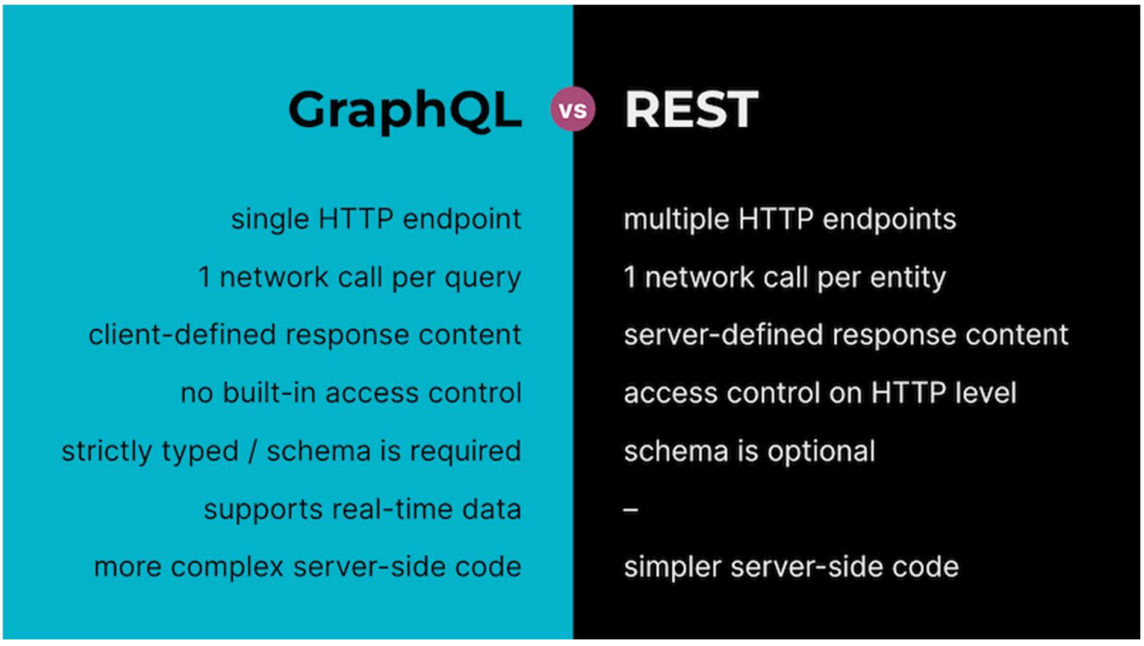
6.1 Factors favoring the selection of GraphQL over REST

The key benefit to consider when making this decision can be found in Table 1Table 2 and is also briefly described in this subchapter. GraphQL allows for efficient data fetching, as clients

can request only the data they need, thereby reducing unnecessary data transfer. It reduces the number of network requests, making data retrieval more efficient by fetching related data in a single query. GraphQL provides flexibility by letting clients define the structure of the response, making it adaptable to changing requirements without impacting existing queries. Versioning is simplified because clients specify their data requirements, reducing the need for maintaining multiple API versions. Real-time capabilities are supported through subscriptions, making it suitable for applications that require live data updates. Strongly typed schemas and introspection offer clear documentation and validation for APIs, aiding developers and tooling. It can aggregate data from multiple sources, making it useful for applications with data distributed across various services or databases.

Front-end developers are empowered by their ability to request data as needed, streamlining development workflows and reducing dependencies on back-end changes. Efficient mobile development is facilitated by GraphQL, reducing data transfer and improving performance for mobile applications. GraphQL is well-suited for handling complex queries and relationships between data, allowing clients to navigate data structures easily.

Table 2 Differences that might influence decision on which API to utilize (Lukashov, 2022)



Choosing GraphQL over REST is favored due to its efficient data fetching, reduced network requests, flexibility, and support for real-time capabilities, making it a versatile option for modern applications.

6.2 Determining When to Prefer REST over GraphQL

When deciding between REST and GraphQL, it's important to consider the trade-offs involved. While GraphQL offers many useful features, it also introduces additional complexity compared to building a REST API. Therefore, opting for REST might be a better choice if the added complexity of GraphQL is unnecessary or difficult to justify.

One scenario where REST is often a good fit is when client pages directly correspond to REST endpoints. This is commonly observed in CRUD (Create, Read, Update, Delete) user interfaces, such as admin panels. With REST, the one-to-one mapping between client pages and endpoints simplifies the development process. (Lukashov, 2022)

Another factor to consider is when the client and server are developed by the same team. In such cases, working on both the client and server in parallel becomes a natural workflow, providing added flexibility. This collaboration between teams eliminates potential communication issues and streamlines the overall development process.

Additionally, if there is no dedicated team available to manage the GraphQL API, it might be wiser to stick with REST. While the benefits of a GraphQL API become apparent over time, building it initially requires a larger investment, especially when dealing with a substantial number of underlying data sources. Embarking on a GraphQL journey without a team experienced in building GraphQL APIs may prove challenging and inefficient.

It is pertinent to carefully evaluate the specific needs and resources of a project before choosing between REST and GraphQL. While GraphQL brings valuable capabilities, REST can be a more suitable option in cases where simplicity, one-to-one mapping, and availability of experienced teams are prioritized.

There are several scenarios where GraphQL is a preferable choice over REST due to its unique advantages. Client pages often have diverse data requirements that don't align well

with REST endpoints. With GraphQL, client developers gain the flexibility to efficiently communicate with the server and avoid the hassle of creating custom REST endpoints for each specific use case.

When the development of the server and client components is handled by different teams, decoupling them can significantly impact the development velocity. This allows the client team to deliver faster, provided that the organization's structure and culture support such collaboration. A good indicator is whether the client and server teams have a shared backlog and aligned priorities.

In systems where clients need access to data from multiple APIs and data sources, creating an API layer becomes a common architectural pattern. This approach reduces the number of connections and dependencies, making the system more robust. GraphQL is particularly suitable for this use case, as it simplifies the process of aggregating and federating data from different sources.

Applications that heavily rely on event-driven updates may benefit from utilizing more specialized protocols like GRPC. However, GraphQL is well-suited for clients with mixed needs, where they require both regular one-off queries and subscriptions to receive real-time updates (also known as standing queries). By considering these factors, you can determine whether adopting GraphQL as your API framework would be advantageous for your specific use case. (Lukashov, 2022)

In Conclusion, it is challenging to determine whether GraphQL is inherently superior to REST. The choice between the two depends on the specific use case and project requirements. Various sources suggest that GraphQL is advantageous due to its faster development and enhanced flexibility in fetching data. However, other sources argue that REST remains a solid standard for designing web APIs, especially for simpler tasks. They caution that GraphQL's increased complexity and reduced constraints might make it more suitable for more intricate operations. The selection between GraphQL and REST will be dictated by the project's specific needs and the preferences of the development team.

7 Summary

The thesis "Exploring GraphQL as an Alternative to REST" delves into the emergence and adoption of GraphQL as a novel approach to web-based APIs, with a particular focus on its potential to replace or complement RESTful architecture. This research work was driven by four key goals:

The first objective of this thesis is to uncover the motives behind the creation and proliferation of GraphQL. It delves into the historical context and motivations behind GraphQL's development, shedding light on its evolution and the problems it aimed to address. A pivotal question explored in this thesis is whether GraphQL truly serves as a robust alternative to REST. It conducts a comparative analysis, considering the strengths and weaknesses of both approaches, aiming to provide a nuanced perspective on their compatibility and potential coexistence in the world of web APIs.

The third goal is to elucidate how GraphQL operates, from the query language itself to its execution on the server. Furthermore, it delves into the specific issues that GraphQL aims to resolve in comparison to REST, such as over-fetching, under-fetching, and the problem of multiple endpoints. Finally, the thesis delves into the intricate domain of authentication and authorization in GraphQL. It dissects the mechanisms and best practices for securing GraphQL APIs, providing insights into how it differs from REST in terms of safeguarding data and resources.

Through comprehensive research, case studies, and practical examples, this thesis aspires to provide a holistic view of GraphQL's role in modern web development, offering readers a well-informed perspective on its development, potential as a REST alternative, operational intricacies, and security considerations. The aim is to equip developers and decision-makers with the knowledge necessary to make informed choices when designing and implementing APIs in the ever-evolving digital landscape.

8 References

ApolloGraph. (2023, March 15th). *Apollo docs*. Retrieved from apollographql.com: https://www.apollographql.com/docs/apollo-server/getting-started

Bezugla, K. (2019, April 19). *GraphQL: Core Features, Architecture, Pros, and Cons*. Retrieved from dzone.com: https://dzone.com/articles/graphql-core-features-architecture-pros-and-cons

Bezugla, K. (2019, April 19). *GraphQL: Core Features, Architecture, Pros, and Cons*. Retrieved from DZone: https://dzone.com/articles/graphql-core-features-architecture-pros-and-cons

Charboneau, T. (2022, March 03). *6 Examples of GraphQL in Production at Large Companies*. Retrieved April 12th, 2022, from NORDIC APIs: https://nordicapis.com/6-examples-of-graphql-in-production-at-large-companies/

Clark, B. (2019, August 13th). *What is GraphQL: History, Components, and Ecosystem*. Retrieved April 12th, 2022, from Level Up Coding: https://levelup.gitconnected.com/what-is-graphql-87fc7687b042

Escott, E. (2020, November 05). *How do you implement CRUD using GraphQL?* Retrieved from CODEBOTS: codebots.com

Gourley, D., & Totty, B. (2002). *HTTP, The Definitive Guide.* Sebastopol: O'Reilly Media. Inc.

Gudabayev, T. (2021, Oct 20). *A Brief History of GraphQL*. Retrieved April 21, 2022, from dev.to: https://dev.to/tamerlang/a-brief-history-of-graphql-2jhd

Gupta, L. (2022, April 07). *REST API Tutorial*. Retrieved from restfulapi.net: https://restfulapi.net/

Hartman, J. (2023, January 28). *GraphQL vs REST*. Retrieved from GURU99: https://www.guru99.com/graphql-vs-rest-apis.html#:~:text=GraphQL%20is%20an%20application%20layer,arranged%20in%20terms%20of%20endpoints.

Hati, S. (2023, January 18th). What is Rest API? Features, Principles, And Challenges.

HTTP. (2023, March 07). Retrieved from httpwg.org: https://httpwg.org/specs/rfc7231.html#method.properties

Husar, A. (2022, April 25). *How to Use REST APIs – A Complete Beginner's Guide*. Retrieved from www.freecodecamp.org: https://www.freecodecamp.org/news/how-to-use-rest-api/

Lukashov, V. (2022, November 4). *stepzen*. Retrieved from stepzen.com: https://stepzen.com/blog/rest-versus-graphql

Masand, A. (2020, October 12th). *buddy.works*. Retrieved from buddy.works: https://buddy.works/tutorials/what-is-graphql-and-why-facebook-felt-the-need-to-build-it

Misquissene, C. J. (2020, April 28). *DEV*. Retrieved from dev.to: https://dev.to/callebdev/understanding-graphql-and-its-design-principles-fd6

mobilelive. (2023, 12 09). *A Mobile Developer's Case for Using GraphQL: Unlocking Efficiency and Flexibility*. Retrieved from medium.com: https://mobilelive.medium.com/a-mobile-developers-case-for-using-graphql-unlocking-efficiency-and-flexibility-43266b7f1ee2

Mychal. (2020, January 13th). *A Brief Tour of GraphQL*. Retrieved from dev.to: https://dev.to/mychal/a-brief-tour-of-graphql-4lcg

Porcello, E., & Banks, A. (2018). *Learning GraphQL (Declarative Data Fetching for Modern Web Apps).* United States: O'Reilly Media, Inc.

Prisma, & GraphQL community. (2023, March 30). *How to GraphQL*. Retrieved from https://www.howtographql.com/: https://www.howtographql.com/basics/2-core-concepts/

Quique. (2020, April 01). *pragmaticreviews*. Retrieved from pragmaticreviews: https://pragmaticreviews.com/what-is-graphql-learn-it-in-5-minutes/

readme. (2016, Nov 15). *readme Blog*. Retrieved from blog.readme.com: https://blog.readme.com/the-history-of-rest-apis/

Richardson, L., & Ruby, S. (2007). *REST Web Services.* O'REILLY.

Roy, F., & Fielding, R. T. (n.d.).

Stubailo, S. (2017, June 27). *GraphQL vs. REST*. Retrieved from APOLLO BLOG: https://www.apollographql.com/blog/graphql/basics/graphql-vs-rest/#url-routes-vs-graphql-schema

The GraphQL Foundation. (n.d.). *GraphQL*. (The GraphQL Foundation) Retrieved April 10th, 2022, from GraphQL: https://graphql.org/

Tomar, A. (2022, Februrary 01). *Create GraphQL API With Example*. Retrieved from Knowledgehut.com: https://www.knowledgehut.com/blog/web-development/create-graphql-api-with-example