The Evolution of API Architectures; REST & GraphQL

by

NICK SWANSON

A THESIS

Presented to the Department of Computer Science and the Robert D. Clark Honors College in partial fulfillment of the requirements for the degree of Bachelor of Science

May 2024

An Abstract of the Thesis of

Nick Swanson for the degree of Bachelor of Science
in the Department of Computer Science to be taken June 2024

Title: The Evolution of API Architectures; REST & GraphQL

Approved:_____*Michal Young Ph.D.*_____
Primary Thesis Advisor

An Application Programming Interface (API) allows two or more applications to communicate with each other and exchange information. The evolution of the internet has led to the development of different API architectures such as Simple Object Access Protocol (SOAP), Representational State Transfer (REST), and GraphQL, each offering their own constraints and benefits. An API architecture is a set of guidelines and principles that define how an API should behave. For example, APIs that follow the constraints of REST are supposed to be flexible and scalable, and APIs built using GraphQL are supposed to be quick and performant. GraphQL is slightly different from REST since it is a query language rather than an architecture. This means that while REST provides a set of constraints that should be followed, GraphQL is an application-level server-side technology that produces a variety of benefits when used. Although REST and GraphQL are different in the way they are implemented, they still serve the same purpose of allowing two or more applications to communicate with each other. The purpose of this paper is to carry out a comparison and analysis between REST and GraphQL, and use that comparison to show how API architectures have evolved over the years. My goal is to provide developers with a case study to show how and why API architectures have evolved since the early 2000's. This will help developers make better informed decisions when building applications and will hopefully open doors for further research in this area.

Table of Contents

Introduction

An Application Programming Interface (API) allows two or more applications to communicate with each other and exchange information. APIs can be designed in different ways, and there is no single rulebook on how APIs should be implemented. There are a few existing architectures such as Simple Object Access Protocol (SOAP), Representational State Transfer (REST), and GraphQL that outline how APIs should be built, but for the most part, there is no clear answer as to which architecture should be used in certain scenarios. Each API architecture is designed to elicit a variety of benefits when used. For example, APIs built following the constraints of REST are supposed to be flexible and scalable:

> REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an internet scale distributed hypermedia system. (R. T. Fielding)

While many API design patterns are referred to as architectures, most are actually architectural styles that outline a set of rules for how APIs should be built: “REST is not an architecture, but rather an architectural style. It is a set of constraints that, when adhered to, will induce a set of properties; most of those properties are believed to be beneficial for decentralized,

network-based applications" (R. T. Fielding). SOAP and REST have been the most popular architectures since their introduction in the early 2000's, however GraphQL has been slowly gaining traction since its introduction in 2015. According to StackShare, over 2,000 companies including Meta, Shopify, and Twitter use GraphQL. IBM also predicts that by 2025, 50% of all enterprises will use GraphQL in production.

GraphQL is different from REST since it is a query language rather than an architectural style. This means that while REST provides a set of constraints that should be followed, GraphQL is an application level server side technology that produces a variety of benefits when building APIs. GraphQL was developed in direct response to issues with REST. During the rise of cell phones and mobile devices, RESTful APIs were experiencing performance issues due to increased network latency. Mobile devices were having trouble rendering HTML due to high network usage relative to the amount of bandwidth available, and one API request could not return all the data needed. Because of this, the client was forced to make multiple redundant API requests to render content. REST also had a fragile client-server relationship, where any changes to the API on the backend needed to be carried over to the frontend. This tight coupling between the frontend and backend created additional work for engineers and made maintaining APIs more difficult. In response to these challenges, a few engineers at Facebook created GraphQL.

GraphQL aims to solve the issue of over fetching data by providing a client driven, declarative query language for APIs. This helps prevent inefficient network requests by allowing the client to state exactly what data they need. This way, the API avoids fetching unnecessary data and only exchanges the exact data requested. This reduces network latency and makes it easier to develop APIs: "GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more,

makes it easier to evolve APIs over time, and enables powerful developer tools" (GraphQL). Behind the scenes, GraphQL tells the server how to present the data to the client. This enables developers to make precise data requests so that they receive exactly what they need - nothing more, nothing less. One can think of GraphQL as having a single 'smart' endpoint rather than many 'dumb' endpoints like REST which returns all data within a specific resource. Because of its superior performance and efficiency, GraphQL is slowly becoming one of the most popular API architectures used in modern web applications.

REST

REST was introduced by Roy Fielding in 2000 with the goal of providing a well defined model of how the components of a web application should interact with each other. Prior to the introduction of REST, Fielding recognized the need for a software architecture that outlined how the World Wide Web should work.

A software architecture is an abstraction of the elements of a software system. It shows how elements are allocated and how they interact with each other. An architectural style places restrictions on elements within a software system, and clearly defines their roles. The core principles of REST as an architectural style stem from the requirements of the World Wide Web.

Timer Berners Lee, the inventor of the World Wide Web, first proposed the idea for a distributed hypermedia system in 1989 while working at the European Council for Nuclear Research (CERN). The key idea of his proposal was Hypertext Transfer Protocol (HTTP). HTTP is a communication protocol that facilitates communication between a client and server. The protocol enables users to access, retrieve, and interact with content hosted on servers around the world through standardized requests and responses using GET, POST, PUT, and DELETE methods. HTTP is an important concept that laid the groundwork for the creation of the World Wide Web.

The World Wide Web was first created as a space where information could be shared and stored:

> What was needed was a way for people to store and structure their own information, whether permanent or ephemeral in nature, such that it could be usable by themselves and

> others, and to be able to reference and structure the information stored by others so that it would not be necessary for everyone to keep and maintain local copies. The intended end-users of this system were located around the world, at various university and government high-energy physics research labs connected via the Internet. (R. T. Fielding)

The goal of the World Wide Web was to build a system that would allow people to share and store structured information. Since participation in this system was voluntary, the creators of the World Wide Web wanted it to have a low barrier of entry to encourage adoption. The creators also wanted the World Wide Web to be extensible, scalable, and distributed. Despite being conceptually sound, the World Wide Web faced many issues early on. Mainly, there was a lack of standardized communication principles and protocols which made interoperability difficult. This made it difficult to establish seamless interactions between different systems. The need for standardized protocols became increasingly apparent, and a few architectures, such as SOAP, emerged in an attempt to fix these problems.

First introduced by Microsoft in 1998, SOAP played an important role in encouraging interoperability among different systems and platforms. At its core, SOAP is a protocol for exchanging structured information in the implementation of web services. SOAP operates by defining a set of rules for structuring messages, allowing for the exchange of information between applications over a network. It relies on Extensible Markup Language (XML) as its message format, providing a standardized and platform-independent means of communication. The main advantage of SOAP is its flexibility in supporting many different communication patterns, including synchronous and asynchronous interactions.

Despite its flexibility, SOAP faced several significant challenges that contributed to its lack of success. One major issue was its complexity. SOAP's XML-based messaging format and extensive specifications made it challenging for developers to work with. The protocol also incurred significant overhead, resulting in large message sizes and slow performance. SOAP also faced compatibility challenges since different implementations sometimes deviated from specifications, leading to interoperability issues.

Fielding recognized these challenges and saw the need for a more cohesive, scalable, and extensible architectural style that could address the shortcomings of existing approaches such as SOAP. Fielding aimed to create an architecture that would be easily understandable, scalable, and adaptable to the evolving nature of the World Wide Web. The motivation behind creating REST was to establish a set of principles that could provide a simple, standardized way for components of the web to communicate.

In its simplest form, REST is an abstraction of the elements in a distributed hypermedia system. More specifically, REST focuses on three classes of elements: processing elements, data elements, and connecting elements. Data elements are the key focus of REST due to the way distributed hypermedia systems function. In a distributed hypermedia system such as the World Wide Web, data needs to be transported from where it is being stored to where it is going to be used. There are three ways this can be achieved:

> A distributed hypermedia architect has only three fundamental options: (1) render the data where it is located and send a fixed-format image to the recipient; (2) encapsulate the data with a rendering engine and send both to the recipient; or (3) send the raw data to

> the recipient along with metadata that describes the data type, so that the recipient can choose their own rendering engine. (R. T. Fielding)

Each option has its advantages and disadvantages. Option one, for example, makes client implementation easy because the client receives a fixed form of the data. This method leads to scalability issues, however, because the server is forced to do a majority of the processing. Option two takes some processing work off of the server, but increases the amount of data being transferred. Option three minimizes the amount of data being transferred but requires the sender and receiver to work with the same data type. By focusing on the data elements in a distributed hypermedia system, REST serves as a hybrid of all three models above by limiting what is revealed to the receiver and using metadata to ensure that both the sender and receiver comprend the data types:

> REST provides a hybrid of all three options by focusing on a shared understanding of data types with metadata, but limiting the scope of what is revealed to a standardized interface. REST components communicate by transferring a representation of the data in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the data. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface. (R. T. Fielding)

The benefit of this model is similar to the benefit in option one where the client receives a fixed form of the data; however, this model also addresses scalability issues and keeps the information

hidden. By transferring a representation of the data, REST captures the current or intended state of resources, simplifies communication between components, and standardizes data formats for interoperability.

Although data elements are the main focus of REST, processing elements and connecting elements play an equally important role in defining REST. There are two main types of processing elements (components) in REST: user agents and origin servers. User agents, such as a web browser, are a type of processing element that use connecting elements to initiate communication with a server. Origin servers use connecting elements to manage resources and receive requests from user agents. This model is crucial for ensuring efficient, reliable, and independent communication.

Connecting elements (connectors) serve as an interface for component communication and hide the underlying details of communication implementation. The main connector type is client-server. In this relationship, the client initiates communication by making a request to the server. The server then handles the request by responding with some information. Connectors are stateless, meaning that the server does not store any information about the client:

> All REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it. This restriction accomplishes four functions: (1) it removes any need for the connectors to retain application state between requests, thus reducing consumption of physical resources and improving scalability; (2) it allows interactions to be processed in parallel without requiring that the processing mechanism understand the interaction semantics; (3) it allows an intermediary to view and understand a request in isolation,

> which may be necessary when services are dynamically rearranged; and (4) it forces all of the information that might factor into the reusability of a cached response to be present in each request. (R. T. Fielding)

Stateless communication simplifies the implementation of both the client and server. Since each request is self-contained, there's no need for the server to store information about the client between requests. Stateless communication also enhances reliability since it eliminates the need for a server to maintain information about a session. This reduces the chance of issues such as session timeouts.

Although REST is stateless, developers can still build stateful applications on top of it through the use of cookies that store session information. This provides flexibility for developers and allows them to create applications that benefit from statefulness. For example, most modern websites that contain login pages benefit from statefulness because it allows the website to keep track of the user's current session and prevents the user from having to login every time they reload the page. The last important feature of connectors is cacheability. Cacheability means that clients can use a cache to avoid repeated network communication, and servers can use a cache to avoid fetching redundant data. This limits repetitive work and improves latency.

To summarize, REST components communicate by transferring a representation of data in a format that matches a specific data type selected based on the needs of the receiver. By transferring a representation of the data, REST is able to capture the current state of the resource and simplify communication between components. The client-server model in REST simplifies component implementation, reduces connector semantic complexity, and enhances performance. The stateless nature of REST also allows each interaction to be independent, removing the need

for components to be aware of each other, and because of this, connectors only need to be aware of each other during communication. The cacheability of connectors further helps improve performance by storing information that would otherwise be repeated work.

Despite being well designed, REST isn't perfect. RESTful systems suffer from over fetching data, lack support for real-time communication, and make API versioning complex. The most significant of these issues is over fetching data. Over fetching occurs when a request returns data that the client is not going to use. In other words, if a client makes a request to a server for a specific piece of data, the client will receive all data that is associated with the endpoint whether they need it or not. For example, if a client requests detailed information about a specific user from a customer database, the server might respond with all available details about that user such as creation date and last login, despite the client only needing the user's name and email. This can lead to severe performance issues because the server has to transfer an unnecessary amount of data, resulting in increased latency and higher resource consumption. This issue becomes particularly problematic when dealing with large datasets or in situations where bandwidth is limited.

The second main issue with REST, lack of real-time communication support, stems from the fact that REST is stateless, meaning that each request-response cycle is independent from all other interactions. This is beneficial because it removes the need for connectors to keep application state between requests, reducing resource consumption and improving scalability, and allows interactions to be processed in parallel. However, this also means that every request can only receive one response and lost responses can't be recovered. This makes real time communication difficult because the client can't maintain a persistent connection with the server. Furthermore, each request is forced to wait for a response which results in high latency. This is

not ideal for services such as instant messaging and modern applications such as social media websites that require real time communication support:

> The rise of instant messaging services, mobile push notifications, and social networking was reflected in the research themes of annual workshops at Irvine from 1998-2000 on event notification, namespaces, and decentralized organizations. As we explored these complementary innovations around the Web, we concluded that real-time, internet-scale event notification - group messaging that can be initiated by any party, at any time - highlighted three limitations of REST's request-response model: One-shot: Every request can only generate a single response. If that response message is an error (or lost), there is no recovery protocol. One-to-one: Every request proceeds from one client to one server. Instead of routing to a group at once, a chain of proxies passes it to each. One-way: Every request must be initiated by a client, and every response must be generated immediately, precluding servers from sending asynchronous notifications. (Roy T. Fielding)

These limitations highlight the challenges in achieving scalable, bidirectional, and asynchronous communication. The drawbacks mentioned are significant because they prevent REST from being a viable architecture in today's world. With the rise of social networking, many applications require real-time communication. REST's request-response model prevents this and forces applications to resort to workarounds like websockets, which can be more complex and resource-intensive.

The last main issue with REST, complex API versioning, poses a significant challenge to developers. As web applications evolve over time they need to be modified and updated with

new features. This includes creating and updating API endpoints. REST lacks a clear standardized approach to API versioning which can lead to compatibility issues between the client and server. For example, if a developer adds a new feature to a web application and updates an API endpoint, existing clients unaware of the update may break. This could disrupt entire applications and lead to increased development time.

Although successful early on, REST is becoming increasingly less viable as the World Wide Web evolves because it suffers from over fetching data, lacks real-time communication support, and makes API versioning complex. These are all issues that are unacceptable in today's world. The majority of the World Wide Web is made up of e-commerce and social networking, both of which require efficient real-time communication. In response to these challenges, alternate approaches such as GraphQL have emerged in an attempt to solve these issues.

GraphQL

Since the World Wide Web has drastically evolved over the past twenty years, GraphQL has emerged as an appealing alternative to REST because it can be more performant and efficient in certain scenarios. Over the past twenty years, there has been a dramatic increase in the number of devices connected to the web. This increase has led to a greater demand for more efficient communication between clients and servers because bandwidth is becoming more and more scarce. Mobile applications, social media platforms, and other online services are constantly transmitting large amounts of data, which strains bandwidth. This issue is becoming more apparent as the number of devices connected to the internet continues to grow: “A forecast by International Data Corporation (IDC) estimates that there will be 41.6 billion IoT devices in 2025, capable of generating 79.4 zettabytes (ZB) of data” (Dell Technologies Info Hub).

During this sudden increase in the number of devices connected to the web, REST was experiencing issues with network latency. Cell phones and mobile devices were having trouble rendering HTML due to high network usage, and one API request could not return all the data needed, so the client was forced to make multiple redundant API requests to render content. Since REST is stateless and over fetches data, each redundant API request would return all of the data located within a specific resource, even if the client only needed a specific piece of the data, and this led to excessive amounts of data being sent between the client and server, significantly straining bandwidth. This strain directly translated into a less responsive user experience. Mobile devices, already restricted by processing power and network capabilities, struggled to handle the influx of unnecessary data. This resulted in slow and unresponsive interfaces, diminishing the

overall quality of the user's experience. In addition to this, RESTful APIs had a fragile client-server relationship and were difficult to maintain. Any changes on the frontend needed to be carried over to the backend.

Recognizing the need for a more performant and efficient solution, a team of engineers at Facebook created GraphQL. GraphQL is a query language used for reading and mutating data in APIs. GraphQL provides developers with a type system where they can describe a schema for their data. In turn, this gives consumers of the API the power to request the exact data they need. A GraphQL API also has a single entry point, meaning that data is fetched by the client who describes the data they need in a syntax that is similar to its return format. This helps solve the issue of inefficient network requests because the client is able to state exactly what data they need and the API avoids fetching unnecessary information and only exchanges the exact data requested. This reduces network latency because less information is being sent from the server to the client: "GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools… Apps using GraphQL are fast and stable because they control the data they get, not the server" (GraphQL). In essence, GraphQL tells the API what data to send to the client. This allows developers to make exact data requests so that they receive exactly what they need - nothing more, nothing less. To give an analogy, GraphQL can be thought of as having a single endpoint that carefully extracts and returns a precise amount of data: "Instead of having multiple endpoints that return fixed data structures, GraphQL APIs typically only expose a single endpoint. This works because the structure of the data that's returned is not fixed. Instead, it's completely flexible and lets the client decide what data is actually needed" (GraphQL). More specifically, clients can query a

server to express its data needs, and the server will respond with the requested data. Queries can range from simple requests such as the names of all users located within a specific database to complex, nested requests. The figure below depicts how GraphQL's syntax differs from REST:

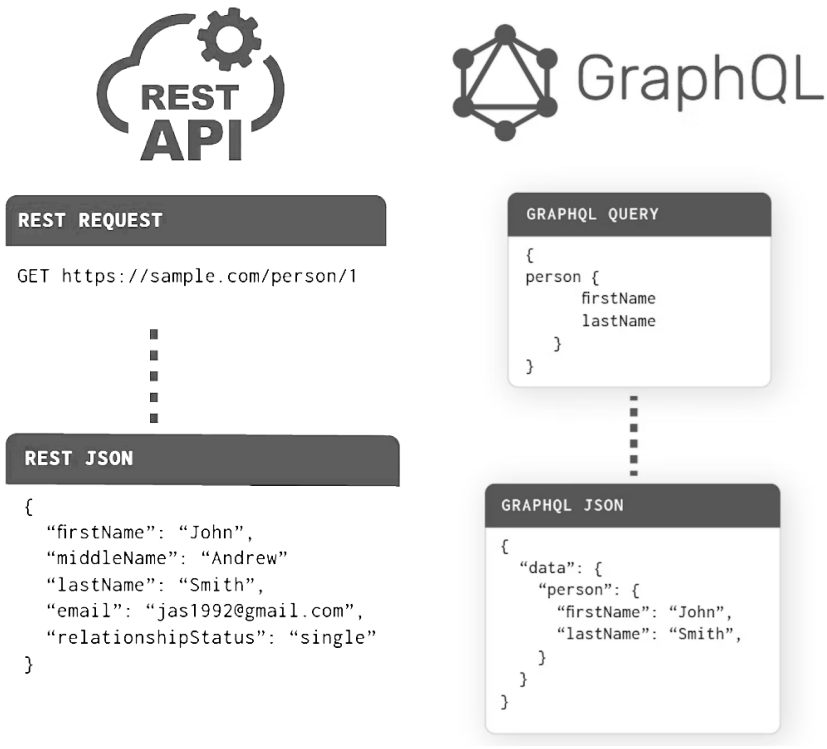

Figure 1: REST vs. GraphQL Request & Response

GraphQL queries are structured in a JSON-like format, and have the exact same shape as the return result. This is important because the client always gets back a predictable data format and the server always knows exactly what the client is requesting. This is also what allows a client to request data that is nested within objects. GraphQL queries allow clients to traverse objects and their fields, making it very easy to access and fetch large amounts of data in one request. In addition, every field in a GraphQL query has its own set of arguments, meaning that the client can further specify what type of data it needs. For example, a field can pass in an

argument such as ‘last’, and the server will only respond with the last item in the database. Arguments also allow clients to transform data: “Every field and nested object can get its own set of arguments, making GraphQL a complete replacement for making multiple API fetches. You can even pass arguments into scalar fields, to implement data transformations once on the server, instead of on every client separately” (GraphQL). This is powerful because clients are able to transform data into their preferred format without taking on too much work. Arguments can also be factored out into variables, making queries more dynamic and scalable. For example, instead of hard coding an argument into a query, the client can factor the argument out into a variable, and the query can fetch certain data depending on the value assigned to the variable.

GraphQL queries also allow clients to mutate data. This means that in addition to reading data, clients can also create, update and delete server-side data. Each of these three actions are their own mutation and serve a different purpose. Mutations have the same syntax as queries, except instead of fetching data from the server, they either create new data on the server, update existing data on the server, or delete existing data on the server. Mutations are also implemented in series so no two separate mutations will ever encounter a race condition:

> A mutation can contain multiple fields, just like a query. There's one important distinction between queries and mutations, other than the name: While query fields are executed in parallel, mutation fields run in series, one after the other. This means that if we send two mutations in one request, the first is guaranteed to finish before the second begins, ensuring that we don't end up with a race condition with ourselves. (GraphQL)

This is a safe and efficient way to allow clients to mutate server-side data. By allowing clients to create, update, and delete data, GraphQL provides all of the functionality that REST does but in a more efficient way.

The last important feature of GraphQL is subscriptions. Subscriptions allow real-time communication between a client and server. Subscriptions work by having a client subscribe to an event, and whenever a change is made, the server sends the update to the client: "When a client subscribes to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client. Unlike queries and mutations that follow a typical "request-response-cycle", subscriptions represent a stream of data sent over to the client" (GraphQL). This is important because subscriptions allow applications to maintain a live connection and receive real-time updates as soon as certain events take place on the server. Real-time communication is an essential feature that many modern applications require and subscriptions are yet another feature that sets GraphQL apart from its REST counterpart.

Despite its advantages over REST, GraphQL also has a few downsides. First, GraphQL can be difficult to learn and introduces another layer of complexity. Although developers have the power to write their own queries, it can be difficult to write larger and more complex queries that accurately do what they are intended to do. This slows down the development process and can cause issues for teams building software applications. Another downside with GraphQL is that caching can be more difficult in certain scenarios compared to REST. Caching is an important feature that can drastically improve performance by reducing redundant requests. While caching is a well-established practice in REST, GraphQL does not have any pre-defined standards for caching. Since GraphQL queries are dynamic, it is difficult to implement caching

because the cached data must match the structure of the individual query. Therefore, developers must implement custom caching logic based on the applications needs. If done poorly, this could lead to performance issues which would negate the purpose of using GraphQL in the first place. Lastly, all GraphQL queries return with an HTTP status code of 200 regardless of whether the query was successful or not. If the query was unsuccessful, the JSON response will return with an error message. This makes it difficult for developers to debug applications and can once again slow down the development process.

Analysis

Despite the drawbacks of GraphQL, it's still a better alternative to REST because it's more efficient and scalable in most scenarios. When considering trade offs, these characteristics are usually far more important than the simplicity that comes with REST. And although only mentioned in theory so far, the performance increase seen in applications that use GraphQL is substantial. Gleison Brito, Thais Mombach, and Marco Tulio Valente's study: "Migrating to GraphQL: A Practical Assessment" focuses on performance boosts seen in applications that use GraphQL. Published in 2019, the paper begins with a literature review to gain an in-depth understanding of the key benefits and characteristics associated with GraphQL. The paper then assesses the practical benefits of GraphQL by migrating seven systems from REST to GraphQL. The findings reveal significant reductions in the size of JSON documents returned by APIs after migration, with reductions ranging from 94% to 99% in terms of the number of fields and bytes. This evidence highlights the performance advantage GraphQL has when it comes to optimizing data transfer efficiency:

> Client-specific queries can lead to a drastic reduction in the size of JSON responses returned by API providers. On the median, in our study, JSON responses have 93.5 fields, against only 5.5 fields after migration to GraphQL, which represents a reduction of 94%. In terms of bytes, we also measure an impressive reduction: from 9.8 MB (REST) to 86 KB (GraphQL). Altogether, our findings suggest that API providers should seriously consider the adoption of GraphQL. (Brito)

By using a type schema, GraphQL was able to reduce the number of fields in JSON responses from 93.5 to 5.5. GraphQL was also able to reduce the number of bytes transferred from 9,800 KB to just 86 KB. These findings show that GraphQL is capable of providing significant performance boosts to real world applications. These results also imply that GraphQL is best suited for applications that handle large amounts of data because there is a larger performance gap between REST and GraphQL as the size of an API request increases.

A similar study conducted by Maximilian Vogel, Sebastian Weber, and Christian Zirpins revealed that migrating an application from REST to GraphQL can also lead to quicker data retrieval times. In their study "Experiences on Migrating RESTful Webservices to GraphQL", the researchers migrated a smart home system from REST to GraphQL and performed testing on the amount of time (in milliseconds) it took each system to retrieve data from a server: "The median was 313 ms for the GraphQL query and 677 ms for consecutive requests using the original REST API. For calling the specific endpoint, the median was 336 ms. Therefore, GraphQL required 46% of the time on average to retrieve all data from the server compared to the REST API" (Vogel). This study reveals that in addition to reducing the number of fields in JSON responses, GraphQL significantly reduces the amount of time that it takes to execute a request. This further shows that GraphQL is more performant than REST and should be used in applications where low latency is valued.

Matheus Seabra, Marcos Felipe Nazário, and Gustavo Pinto's study: "REST or GraphQL? A Performance Comparative Study" also investigates the performance differences between REST and GraphQL. Their research evaluates three applications implemented using either REST or GraphQL and compares key performance metrics such as response time and data transfer rate. Their findings also indicate that migrating to GraphQL leads to improved

performance in terms of the average number of requests per second and the transfer rate of data: “It was observed that migrating to GraphQL resulted in an increase in performance in two-thirds of the tested applications, with respect to average number of requests per second and transfer rate of data” (Matheus). Similar to the results found in “Migrating to GraphQL: A Practical Assessment”, the results found in this study show that GraphQL reduces the number of requests per second and the number of bytes transferred between the client and server which leads to better data transfer efficiency. As mentioned before, this is due to the fact that GraphQ allows the client to request and receive only the exact data they need.

Lastly, although research shows that GraphQL is more performant than REST, studies show that it can also be easier to implement. In order to determine whether GraphQL is a viable alternative to REST, Sri Vadlamani, Benjamin Emdon, Joshua Arts, and Olga Baysal, carried out a qualitative study in their paper “Can GraphQL Replace Rest? A Study of Their Efficiency and Viability”. In their study, two APIs were qualitatively analyzed by surveying thirty-eight engineers at GitHub Inc. The goal of the survey was to understand developer perceptions about REST and GraphQL. The survey included questions about the perceived strengths and weaknesses of REST and GraphQL, the efficiency of GraphQL in requesting resources, and their expectations for the adoption of GraphQL over the next five years. According to the results, a majority of the engineers at GitHub believe that it is easier to efficiently request resources with GraphQL: “A majority (55%) of participants agreed that GraphQL APIs make it easier to efficiently request resources. On the other hand, about 36% of participants remained neutral” (S. L. Vadlamani). This is important because one of the most significant downsides of GraphQL is that it can be difficult to learn and implement. However, this study shows that GraphQL is not necessarily more difficult to use and implement compared to REST.

Overall, the first two studies show that GraphQL offers significant performance advantages over REST, particularly for applications that handle large amounts of data. The last study also shows that GraphQL can be easier to implement in certain scenarios compared to REST. Therefore, it's clear that GraphQL should be used over REST in most modern applications because it is more performant, scalable, and easier to work with.

Discussion

The purpose of this paper was to carry out a comparison and analysis between REST and GraphQL, and use that comparison to show how API architectures have evolved over the years. When the World Wide Web was first created, there was very little structure. Roy Fielding quickly realized this and recognized the need for a software architecture that outlined how the World Wide Web should work. Because of this, he introduced REST in 2000 with the goal of providing a well defined model of how components inside of a web application should interact with each other: “REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations” (R. T. Fielding). The goal of REST was to minimize network communication and maximize independence. This was appropriate at the time because many applications did not require real-time communication and the amount of data being transferred between a client and server was minimal. However, since the internet has drastically evolved since 2000, many modern applications, such as social media platforms and other online services, need the ability to maintain live connections with their clients. The amount of information being transferred between clients and servers is also drastically more than it was twenty-four years ago. Hence, REST has become increasingly less viable because it suffers from over fetching data and lacks real-time communication support. In light of this, GraphQL, a query language used for reading and mutating data in APIs, has emerged as an appealing alternative to REST because it is more performant and efficient than REST, and supports real-time communication through the use of subscriptions. Research shows that GraphQL reduces the size of JSON response documents by 94% to 99% in terms of the number of fields and bytes. Research also shows that GraphQL

significantly reduces the number of bytes transferred between a client and server, provides quicker data retrieval times, and reduces the number of requests sent per second. These findings highlight the evolution of API architectures. GraphQL can be thought of as an evolution of REST. Similar to REST, GraphQL is a stateless protocol used for client-server communication. However, GraphQL has evolved to meet the demands of modern applications by providing a more efficient way to query and manipulate data. Therefore, it's clear that GraphQL should be used over REST in many modern applications. There are a few exceptions however. If performance is not a priority, or if you intend working with a relatively small amount of data, REST might be the better option because it is generally easier to work with and takes less time to implement. However, if you plan on working with a large amount of data and value performance, GraphQL is the clear winner.

Bibliography

Brito Gleison, Thais Mombach, and Marco Tulio Valente. “Migrating to GraphQL: A Practical Assessment.” 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019.

Brito Gleison, and Marco Tulio Valente. “REST vs GraphQL: A controlled experiment.” 2020 IEEE international conference on software architecture (ICSA). IEEE, 2020.

“Internet of Things and Data Placement: Edge to Core and the Internet of Things.” *Dell Technologies Info Hub*, infohub.delltechnologies.com/en-us/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement/. Accessed 18 Apr. 2024.

GraphQL, graphql.org/. Accessed 16 Mar. 2024.

“GraphQL Core Concepts.” *GraphQL Core Concepts*, www.howtographql.com/basics/2-core-concepts/. Accessed 16 Mar. 2024.

Leonard Richardson and Sam Ruby. “RESTful Web Services: Web services for the real world”. O’Reilly Media, 2007.

Matheus Seabra, Marcos Felipe Nazário, and Gustavo Pinto. 2019. “REST or GraphQL? A Performance Comparative Study”. In Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19). Association for Computing Machinery.

Meagher, Kyle. “GraphQL vs. Rest: A Quick Guide.” *Cosmic*, 16 Feb. 2021, www.cosmicjs.com/blog/graphql-vs-rest-a-quick-guide.

R. T. Fielding and R. N. Taylor, “Principled design of the modern Web architecture,” Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium, Limerick, Ireland, 2000, pp. 407-416, doi: 10.1145/337180.337228.

Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. “Reflections on the REST architectural style and principled design of the modern web architecture” (impact paper award). In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017).

S. L. Vadlamani, B. Emdon, J. Arts and O. Baysal, “Can GraphQL Replace REST? A Study of Their Efficiency and Viability,” 2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), Madrid, Spain, 2021, pp. 10-17, doi: 10.1109/SER-IP52554.2021.00009.

Vogel, M., Weber, S., Zirpins, C. (2018). “Experiences on Migrating RESTful Web Services to GraphQL”. In: Braubach, L., *et al.* Service-Oriented Computing – ICSOC 2017 Workshops. ICSOC 2017. Lecture Notes in Computer Science(), vol 10797. Springer, Cham. https://doi.org/10.1007/978-3-319-91764-1_23