Demystifying use cases of GraphQL, REST and gRPC

Manaswini Das
Red Hat
dasmanaswini10@gmail.com

Nancy Chauhan
Grofers
nancychn1@gmail.com

ABSTRACT
In API development, we have come a long way since CORBA(Common Object Request Broker Architecture), which was based on RPC(Remote Procedure Calls). We moved on to more flexible solutions to thrust the ease of development like GraphQL and gRPC, which is now rapidly gaining popularity. As a consequence, this led to a lot of comparisons between the existing API styles. We are going to burst the existing myths around API styles. We will discuss the specific use cases with suitable examples and demonstrate what GraphQL, gRPC and REST are made for, and how we can leverage each of them to our advantage based upon the constraints and the type of application we intend to create.

AUDIENCE
This talk is appropriate for both beginner and intermediate level developers.

INTRODUCTION
The API landscape is continuously shifting. Going back to history, with the rise of networks in the 1990s, APIs became distributed across multiple systems, and we created patterns, standards, and frameworks to make building those systems easier. In the 2000s, when XML (Extensible Markup Language) ruled the world, most developers had to deal with SOAP (Simple Object Access Protocol) to integrate APIs. Then the web-friendly REST(Representational State Transfer) and JSON(JavaScript Object Notation) appeared. A lot of arguments around REST and SOAP started. However, with much attention, moving away from SOAP-based Web Services to REST and HTTP(HyperText Transfer Protocol), the debates and discussions died down. Many SOA practitioners adopted REST (or plain HTTP) as the basis for their distributed systems.

Today, GraphQL, gRPC, REST, and WebHooks are among a bewildering array of technologies and architectural styles available to API developers.

REST is an API protocol that was introduced in a 2000 dissertation by Roy Fielding, whose goal was to solve some of the shortcomings of SOAP. REST is more flexible in that it supports a variety of data formats.

On the other hand, gRPC is an open-source API that also falls within the category of RPC. Unlike SOAP, however, gRPC is much newer, having been released publicly by Google in 2015. (That said, the history of gRPC dates back to an internal project at Google called Protocol Buffers(protobufs) that started in 2001.) Like REST and SOAP, gRPC uses HTTP as its transport layer.

GraphQL is an API protocol that was developed internally by Facebook in 2013, then released publicly in 2015.

Broadly, we can classify API styles into five categories, namely:

- Query APIs(for reading data)

- Flat file APIs(used in EDI(Electronic Data Interchange))
- Streaming APIs(which are dynamic and bi-directional, mostly used in OTT(Over THe Top) platforms
- RPC APIs
- Web APIs

API styles provide a way to choose APIs, not the other way around.

When it comes to zeroing on the APIs for your application, we have to consider various constraints. But before that, we firmly believe that the priority should be the properties of the system like scalability, security, features, and then move on to the constraints. GraphQL is a contract built over schema, gRPC over Protobuf, and REST over open API.
Protocol buffers are lightweight compared to JSON or XML, making the systems consume less CPU and network bandwidth. Again, protocol buffers are compressed to achieve the above advantages, which means they are not as human-readable in the compressed form as compared to JSON/XML.

Talking about the various myths around the above, the first one is using REST leads to over-fetching or under-fetching of data. It is true, but there are ways to overcome over-fetching like sparse fieldsets or field grouping, and we have compound documents for under-fetching. The next one being, REST has versioning. We agree to it, but we can't ignore the fact that versioning is a strategy to manage breaking changes and not a requirement. GraphQL, on the other hand, eliminates versioning, as managing different versions can be expensive, making introducing deprecations easier. We agree that we should always prefer graceful evolution, but then it has its constraints. Also, we know that HATEOAS(Hypermedia as the Engine of Application State) is a constraint of the REST application architecture. Even if this constraint is considered a hindrance since it is harder to implement, reduces flexibility, needs more resources, and, most importantly, it isn't easy to achieve deep linking. Still, we can't ignore that it is essential for ease of use and authorization, better maintainability, and tracking.

Coming to the myths around GraphQL, we all know that the one disadvantage that everyone talks about GraphQL is it breaks caching. But before we move on to caching, we must understand the different kinds of caching. Caching may be of three types:

- Client-side/Browser caching
- Server-side/Application caching
- Network side/ Intermediate caching

Now, we know that HTTP caching requires intermediate caching proxy between client and server caching, but GraphQL and RPC don’t use this proxy. Again, we should take faster data delivery into account here. If one gets the data closer to the client, one can choose to localize data and in the process, get rid of expensive calls. We also have to consider API authentication here. Intermediate cache proxy will not cache authenticated traffic by default. We also know that GraphQL APIs contain both queries and mutations which makes it highly customizable. We can not ignore the fact that the more customizable the API is, the less cacheable it will be. So, if network caching is valuable, REST is to be preferred. Also, we know that with GraphQL, there is a single round trip to the server and we can get exactly what we need by means of schema stitching, which is considered to be an advantage of GraphQL, but, we cannot ignore the fact that schema stitching adds complexity.

Coming to performance, with HTTP/1.x, the cost of handshakes was high, while HTTP/2.x removes the need for compound documents since they ruin the cacheability of requests. gRPC uses HTTP/2 extensively for streaming methods.

Again it is said that domain modelling is purely a REST concern, but we think that domain modeling should be done keeping the user in mind. Both gRPC and REST start from an outside-in approach(i.e., the server decides the shape of the queries and then builds it accordingly). GraphQL comes handy when all operations/interactions are not known since it delays the last moment identification of the user needs to profile queries. Also, if the system is capable of consuming flat files, gRPC can be used.

So, summing up, if we are to define the various use cases among REST, GraphQL and gRPC, if we intend to create application web forms, for instance, a ticketing system which makes use of simple CRUD(Create, Read, Update and Delete), REST is the go-to option. If our application makes extensive use of composite APIs or microservices, and we need precise data with ease, GraphQL is the choice since it is data source agnostic. It provides schema stitching, which comes handy when we want to query data from multiple APIs. If we're going to create native mobile BFF(Backend for Frontend) or polyglot microservices, where there is a single client and API is static, well-understood, and owned by the client, we can choose gRPC. gRPC uses Protobuf, which can come handy when we are developing for less performant devices with less CPU power like mobile devices.

We tried to build POCs for various use cases. We came up with specific observations that include feasibility and ease of development to justify all the information discussed above, which we intend to demonstrate.

OUTCOMES/CONCLUSION

Thus, we cannot just go by the words blindly when it comes to following API architectures and styles for our applications. Every approach is unique and has been built for a different cause. We should try to make systems evolvable and simultaneously pay attention to interfaces. We intend to thrust that everything has its pros and cons, but what we have to choose for our application rests in our hands, and should be purely dependent upon the application that we want to build and the properties of the same. The attendees will come to know about the existing API architectural styles and their evolution, followed by some myths around them. We intend to enlighten the attendees with the exact use cases of REST, GraphQL, and gRPC and thus aid in solving the dilemma around API development.

PARTICIPATION STATEMENT

We hereby make a commitment to attend the conference if accepted.

BIO

Manaswini Das is from Bhubaneswar and currently working with the Red Hat Middleware team in Bangalore, India. She is an active open-source user and contributor since 2017. She is an Outreachy alumnus and a Processing Foundation fellow. She loves mentoring, exploring new trends in technology, taking part in conferences, and sketching. She has speaking experience with DjangoCon and PyCon.

Nancy Chauhan is a Software Engineer at Grofers, India. She works on various tech ranging from Backend to DevOps. She has also been a Google Summer of Code Intern in 2019. She likes to teach and mentor about technology. In her free time, she loves to draw or indulge in photography. She has speaking experience with Women Tech Global conference 2020.

REFERENCES/BIBLIOGRAPHY

[1] https://pulsejet.github.io/blog/posts/hateoas/
[2] https://www.youtube.com/watch?v=LIaekiT6Ehs