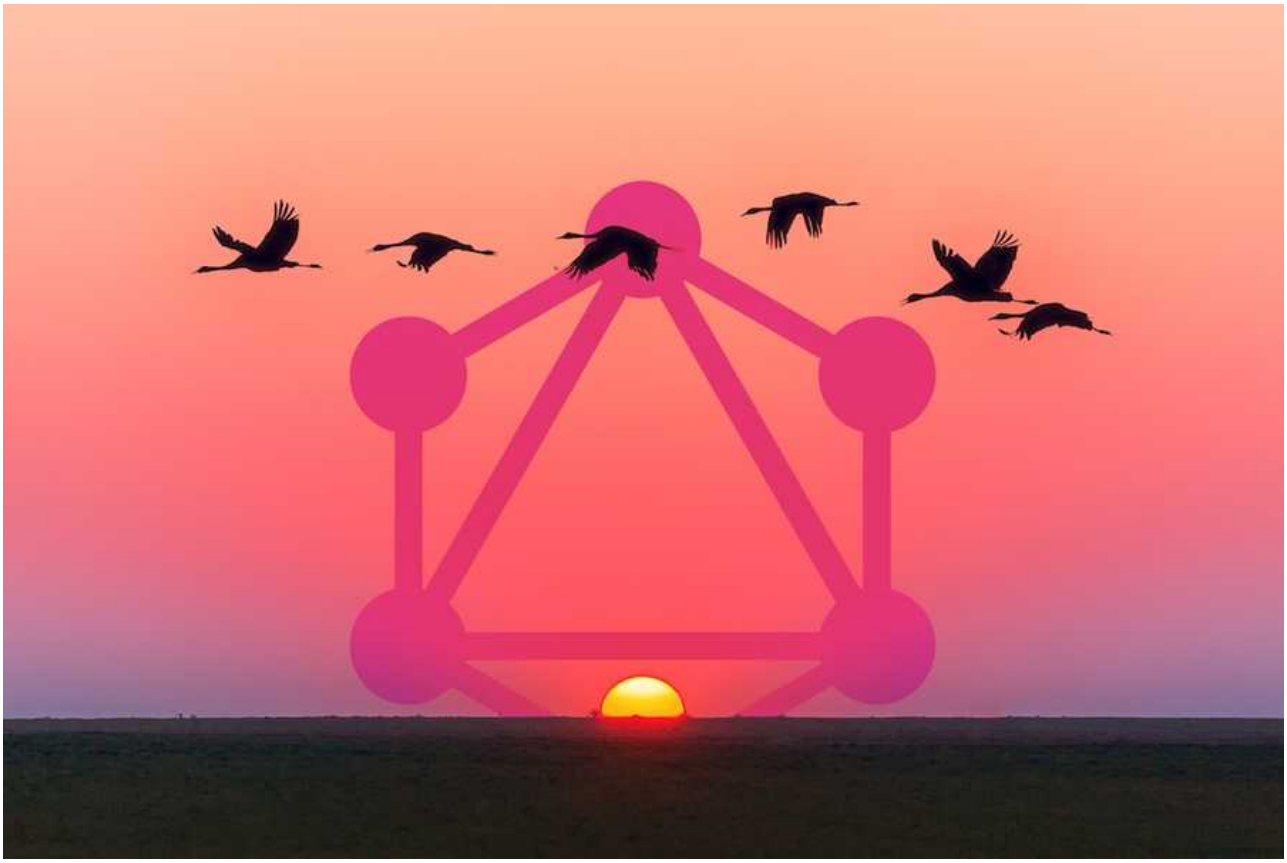
Why GraphQL: Advantages and Disadvantages

JULY 03, 2018 BY ROBIN WIERUCH - EDIT THIS POST



Interested in reading this tutorial as one of many chapters in my GraphQL book? Checkout the entire The Road to GraphQL book that teaches you to become a fullstack developer with JavaScript.

This tutorial is part 1 of 2 in this series.

Part 2 Why Apollo: Advantages, Disadvantages & Alternatives

When it comes to network requests between client and server applications, REST is one of the most popular choices to connect both worlds. In REST, everything evolves around the idea of having resources that are accessible by URLs. You can read a resource with a HTTP GET request, create a resource with a HTTP POST request, and update or delete it with HTTP PUT and DELETE requests. These are called CRUD (Create, Read, Update, Delete) operations.

Resources can be anything from authors, articles, or users. The format for transferring data is not opinionated with REST, but most often people will use JSON for it. In the end, REST enables applications to communicate with each other by using plain HTTP with URLs and HTTP methods.

```
// a RESTful request with HTTP GET
https://api.domain.com/authors/7

// the response in JSON
{
  "id": "7",
  "name": "Robin Wieruch",
  "avatarUrl": "https://domain.com/authors/7",
  "firstName": "Robin",
  "lastName": "Wieruch"
}
```

Though REST was the status quo for a long time, a Facebook technology called GraphQL has recently emerged as a potential successor. The following sections introduce GraphQL's advantages and disadvantages, as well as possible alternatives for developers who need options.

TABLE OF CONTENTS

WHAT IS GRAPHQL?

In short, GraphQL is an open source **query language** created by Facebook, a company that unsurprisingly remains at the pinnacle of web-based software development. Before GraphQL went open source in 2015, Facebook used it internally for their mobile applications since 2012, as an alternative to the common REST architecture. It allows requests for specific data, giving clients more control over what information is sent. This is more difficult with a RESTful architecture because the backend defines what data is available for each resource on each URL, while the frontend always has to request all the information in a resource, even if only a part of it is needed. This problem is called overfetching. In the worst case scenario, a client application has to read multiple resources through multiple network requests. This is overfetching, but also adds the need for waterfall network requests. A query language like GraphQL on the server-side and client-side lets the client decide which data it needs by making a single request to the server. Network usage was reduced dramatically for Facebook's mobile applications as a result, because GraphQL made it more efficient with data transfers.

Facebook open-sourced the GraphQL specification and its reference implementation in JavaScript, and multiple major programming languages implemented the specification since then. The ecosystem around GraphQL is growing horizontally by offering multiple programming languages, but also vertically, with libraries on top of GraphQL like Apollo and Relay.

A GraphQL operation is either a query (read), mutation (write), or subscription (continuous read). Each of those operations is only a string that needs to be constructed according to the GraphQL query language specification. Fortunately, GraphQL is evolving all the time, so there may be other operations in the future.

Once this GraphQL operation reaches the backend application, it can be interpreted against the entire GraphQL schema there, and resolved with data for the frontend application. GraphQL is not opinionated about the network layer, which is often HTTP, nor about the payload format, which is usually JSON. It isn't opinionated about the application architecture

payload format, which is usually JSON. It isn't opinionated about the application architecture at all. It is only a query language.

```
// a GraphQL query
author(id: "7") {
  id
  name
  avatarUrl
  articles(limit: 2) {
    name
    urlSlug
  }
}

// a GraphQL query result
{
  "data": {
    "author": {
      "id": "7",
      "name": "Robin Wieruch",
      "avatarUrl": "https://domain.com/authors/7",
      "articles": [
        {
          "name": "The Road to learn React",
          "urlSlug": "the-road-to-learn-react"
        },
        {
          "name": "React Testing Tutorial",
          "urlSlug": "react-testing-tutorial"
        }
      ]
    }
  }
}
```

One query already requests multiple resources (author, article), called fields in GraphQL, and only a particular set of nested fields for these fields (name, urlSlug for article), even though the entity itself offers more data in its GraphQL schema (e.g. description, releaseData for article). A RESTful architecture needs at least two waterfall requests to retrieve the author entity and its articles, but the GraphQL query made it happen in one. In addition, the query only selected the necessary fields instead of the whole entity.

That's GraphQL in a nutshell. The server application offers a GraphQL schema, where it defines all available data with its hierarchy and types, and a client application only queries the required data.

GRAPHQL ADVANTAGES

GRAPHQL ADVANTAGES

The following list shows the major advantages of using GraphQL in an application.

Declarative Data Fetching

As you've seen, GraphQL embraces declarative data fetching with its queries. The client selects data along with its entities with fields across relationships in one query request. GraphQL decides which fields are needed for its UI, and it almost acts as UI-driven data fetching, like how Airbnb uses it. A search page at Airbnb usually has a search result for homes, experiences, and other domain-specific things. To retrieve all data in one request, a GraphQL query that selects only the part of the data for the UI makes perfect sense. It offers a great separation of concerns: a client knows about the data requirements; the server knows about the data structure and how to resolve the data from a data source (e.g. database, microservice, third-party API).

No Overfetching with GraphQL



There is no overfetching in GraphQL. A mobile client usually overfetches data when there is an identical API as the web client with a RESTful API. With GraphQL, the mobile client can choose a different set of fields, so it can fetch only the information needed for what's onscreen.

GraphQL for React, Angular, Node and Co.

GraphQL is not just exciting for React developers, though. While Facebook showcased GraphQL on a client-side application with React, it is decoupled from any frontend or backend solution. The reference implementation of GraphQL is written in JavaScript, so the usage of GraphQL in Angular, Vue, Express, Hapi, Koa and other JavaScript libraries on the client-side and server-side is possible, and that's just the JavaScript ecosystem. GraphQL does mimic REST's programming language-agnostic interface between two entities, such as client or server.

Who is using GraphQL?

Facebook is the driving company behind the GraphQL specification and reference implementation in JavaScript, but other well-known companies are also using it for their applications. They are invested in the GraphQL ecosystem due to the huge demand for modern applications. Beyond Facebook, GraphQL has also been used by these well-known companies:

- GitHub [1] [2]

- GitHub [1] [2]
- Shopify [1] [2]

- Twitter
- Coursera
- Yelp
- Wordpress
- The New York Times
- Samsara
- and more ...

When GraphQL was developed and open sourced by Facebook, other companies ran into similar issues for their mobile applications. That's how Netflix came up with Falcor, an alternative to GraphQL. It shows again that modern applications demanded solutions like GraphQL and Falcor.

Single Source of Truth



The GraphQL schema is the single source of truth in GraphQL applications. It provides a central location, where all available data is described. The GraphQL schema is usually defined on server-side, but clients can read (query) and write (mutation) data based on the schema. Essentially, the server-side application offers all information about what is available on its side, and the client-side application asks for part of it by performing GraphQL queries, or alters part of it using GraphQL mutations.

GraphQL embraces modern Trends

GraphQL embraces modern trends on how applications are built. You may only have one backend application, but multiple clients on the web, phones, and smartwatches depending on its data. GraphQL can be used to connect both worlds, but also to fulfil the requirements of each client application--network usage requirements, nested relationships of data, fetching only the required data--without a dedicated API for each client. On the server side, there might be one backend, but also a group of microservices that offer their specific functionalities. This defines the perfect use for GraphQL schema stitching, which lets you aggregate all functionalities into one GraphQL schema.

GraphQL Schema Stitching

Schema stitching makes it possible to create one schema out of multiple schemas. Think about a microservices architecture for your backend where each microservice handles the business logic and data for a specific domain. In this case, each microservice can define its own GraphQL schema, after which you'd use schema stitching to weave them into one that is

accessed by the client. Each microservice can have its own GraphQL endpoint, where one GraphQL API gateway consolidates all schemas into one global schema.

GraphQL Introspection

A GraphQL introspection makes it possible to retrieve the GraphQL schema from a GraphQL API. Since the schema has all the information about data available through the GraphQL API, it is perfect for autogenerating API documentation. It can also be used to mock the GraphQL schema client-side, for testing or retrieving schemas from multiple microservices during schema stitching.

Strongly Typed GraphQL

GraphQL is a strongly typed query language because it is written in the expressive GraphQL Schema Definition Language (SDL). Being strongly-typed makes GraphQL less error prone, can be validated during compile-time and can be used for supportive IDE/editor integrations such as auto-completion and validation.

GraphQL Versioning

In GraphQL there are no API versions as there used to be in REST. In REST it is normal to offer multiple versions of an API (e.g. api.domain.com/v1/, api.domain.com/v2/), because the resources or the structure of the resources may change over time. In GraphQL it is possible to deprecate the API on a field level. Thus a client receives a deprecation warning when querying a deprecated field. After a while, the deprecated field may be removed from the schema when not many clients are using it anymore. This makes it possible to evolve a GraphQL API over time without the need for versioning.

A growing GraphQL Ecosystem

The GraphQL ecosystem is growing. There are not only integrations for the strongly typed nature of GraphQL for editors and IDEs, but also standalone applications for GraphQL itself. What you may remember as Postman for REST APIs is now GraphiQL or GraphQL Playground for GraphQL APIs. There are various libraries like Gatsby.js, a static website generator for React using GraphQL. With Gatsby.js, you can build a blog engine by providing your blog content at build-time with a GraphQL API, and you have headless content management systems (CMS) (e.g. GraphCMS) for providing (blog) content with a GraphQL API. More than just technical aspects are evolving; there are conferences, meetups, and communities forming for GraphQL, as well as newsletters and podcasts.

Should I go all in GraphQL?

Adopting GraphQL for an existing tech stack is not an "all-in" process. Migrating from a

Adopting GraphQL for an existing tech stack is not an "all-in" process. Migrating from a monolithic backend application to a microservice architecture is the perfect time to offer a

GraphQL API for new microservices. With multiple microservices, teams can introduce a GraphQL gateway with schema stitching to consolidate a global schema. The API gateway is also used for the monolithic REST application. That's how APIs are bundled into one gateway and migrated to GraphQL.

GRAPHQL DISADVANTAGES

The following topics show you some of the disadvantages of using GraphQL.

GraphQL Query Complexity

People often mistake GraphQL as a replacement for server-side databases, but it's just a query language. Once a query needs to be resolved with data on the server, a GraphQL agnostic implementation usually performs database access. GraphQL isn't opinionated about that. Also, GraphQL doesn't take away performance bottlenecks when you have to access multiple fields (authors, articles, comments) in one query. Whether the request was made in a RESTful architecture or GraphQL, the varied resources and fields still have to be retrieved from a data source. As a result, problems arise when a client requests too many nested fields at once. Frontend developers are not always aware of the work a server-side application has to perform to retrieve data, so there must be a mechanism like maximum query depths, query complexity weighting, avoiding recursion, or persistent queries for stopping inefficient requests from the other side.

GraphQL Rate Limiting

Another problem is rate limiting. Whereas in REST it is simpler to say "we allow only so many resource requests in one day", it becomes difficult to make such a statement for individual GraphQL operations, because it can be everything between a cheap or expensive operation. That's where companies with public GraphQL APIs come up with their specific rate limiting calculations which often boil down to the previously mentioned maximum query depths and query complexity weighting.

GraphQL Caching

Implementing a simplified cache with GraphQL is more complex than implementing it in REST. In REST, resources are accessed with URLs, so you can cache on a resource level because you have the resource URL as identifier. In GraphQL, this becomes complex because

each query can be different, even though it operates on the same entity. You may only request just the name of an author in one query, but want to know the email address in the next. That's where you need a more fine-grained cache at field level, which can be difficult to implement. However, most of the libraries built on top of GraphQL offer caching mechanisms out of the box.

WHY NOT REST?

GraphQL is an alternative to the commonly used RESTful architecture that connects client and server applications. Since REST comes with a URL for each resource, it often leads to inefficient waterfall requests. For instance, imagine you want to fetch an author entity identified by an id, and then you fetch all the articles by this author using the author's id. In GraphQL, this is a single request, which is more efficient. If you only want to fetch the author's articles without the whole author entity, GraphQL lets you to select only the parts you need. In REST, you would overfetch the entire author entity.



Today, client applications are not made for RESTful server applications. The search result on Airbnb's platform shows homes, experiences, and other related things. Homes and experiences would already be their own RESTful resources, so in REST you would have to execute multiple network requests. Using a GraphQL API instead, you can request all entities in one GraphQL query, which can request entities side by side (e.g. homes and experiences) or in nested relationships (e.g. articles of authors). GraphQL shifts the perspective to the client, which decides on the data it needs rather than the server. This is the primary reason GraphQL was invented in the first place, because a Facebook's mobile client required different data than their web client.

There are still cases where REST is a valuable approach for connecting client and server applications, though. Applications are often resource-driven and don't need a flexible query language like GraphQL. However, I recommend you to give GraphQL a shot when developing your next client server architecture to see if it fits your needs.

GRAPHQL ALTERNATIVES

REST is the most popular alternative for GraphQL, as it is still the most common architecture for connecting client and server applications. It became more popular than networking technologies like RPC and SOAP because it used the native features of HTTP, where other protocols like SOAP tried to build their own solution on top of it.

protocols like SOAP tried to build their own solution on top of it.

Falcor by Netflix is another alternative, and it was developed at the same time as GraphQL. Netflix ran into similar issues as Facebook, and eventually open-sourced their own solution. There isn't too much traction around Falcor, maybe because GraphQL got so popular, but developers at Netflix have shown great engineering efforts in the past, so it may be worth looking into it.

. . .

There are plenty of reasons to adopt GraphQL for your JavaScript applications instead of implementing yet another RESTful architecture. It has many advantages, and plays nicely with modern software architecture. This book will introduce how it can be used for many practical, real-life solutions, so you should have an idea if it works for you by the time you've read through the chapters.

This tutorial is part 1 of 2 in this series.

Part 2 Why Apollo: Advantages, Disadvantages & Alternatives

KEEP READING ABOUT FIREBASE ›

HOW TO USE FIREBASE REALTIME DATABASE IN REACT

Now we've worked with a list of data and single entities with the Firebase's realtime database to create an admin dashboard in the previous sections. In this section, I want to introduce a new entity...

REACT WITH APOLLO AND GRAPHQL TUTORIAL

In this tutorial, you will learn how to combine React with GraphQL in your application using Apollo. The Apollo toolset can be used to create a GraphQL client, GraphQL server, and other complementary...

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development
- ✔ Learn JavaScript
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.



PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me