GraphQL

PUBLISH ON28 OCT, 2021 - by Budhaditya Bhattacharya

LAST UPDATED: 06 MAR, 2024

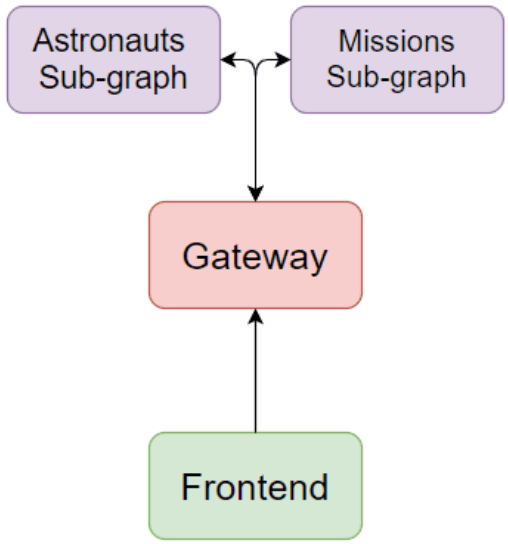
Microservices, the latest iteration of service-oriented architecture (SOA), are the biggest trend in the software development industry, and GraphQL is becoming the preferred query language due to its flexibility. But microservices can be difficult to work with. For example, how do you avoid multiple endpoints for users? One solution is to implement federation.

What is GraphQL federation?

Federated architecture brings different services together into one API endpoint. As an example, imagine you have an application that gives you an overview of Apollo missions and their crew members.

In a typical monolith scenario, you would have a single application that would give you all the needed information about the different missions and crew members. Instead, what you would want is one service to manage information about missions and another service to manage information about astronauts. In this case, you could easily split your monolith into two GraphQL instances. However, now, your frontend would no longer work because it would need to call two different APIs then stitch the information together.

GraphQL federation allows you to set up a single GraphQL API, or a gateway, that fetches from all your other APIs. Your mission service and your astronaut service are now *subgraphs*.



Your frontend only needs to query the gateway, no matter how the services are split up behind it. You can also progressively split up your monolith without changing the frontend. So you can keep the logic for missions in the monolith, for instance, and split out the logic for astronauts. Your frontend queries the gateway, and the gateway queries either the monolith or the astronaut service, depending on what's needed.

What is GraphQL used for?

The list of GraphQL use cases is long and varied. With the right implementation architecture, you can use it to gain insights into how your business operates and to build services and products faster. You can use a developer-centric approach to put APIs at the heart of your business.

Using a tool like Tyk's Universal Data Graph, you can do all of this while enjoying fantastic observability for all your services. Universal Data Graph provides a single, scalable graph that effectively combines all of your services and allows you to query it like a database.

GraphQL is a query language for APIs, but its potential is so much bigger than you might imagine. It can expose all of your services using a single, unified API, even when your services have different protocols, response types and specifications. GraphQL schemas make it easy to communicate how to use the API across stakeholders, with an enhanced developer experience compared to REST APIs.

At the core of GraphQL's flexibility is the fact that it decouples specific use cases from a concrete implementation. As such, you can create a dynamic API that serves many

consumers with different requirements. They can modify a query to suit their needs, with no need to ask the API provider to add or modify a resource, as is the case with REST APIs.

Platforms such as Shopify and GitHub are other good examples of what GraphQL is used for, as public APIs like this, with large groups of unknown users, need GraphQL's flexibility in terms of satisfying so many different use cases. How Netflix scales its API with GraphQL federation is another strong example of GraphQL in action.

Another point in GraphQL's favour is that it encourages best practices such as thinking about API design prior to implementation, strong typing and living documentation.

How does GraphQL work?

GraphQL works by giving clients precisely the data that they request. Nothing more, nothing less. This means you can wave goodbye to the problems of over- and under-fetching. It does this through the use of schemas, which specify the types of objects that clients can request when they query the service. Developers define the object types in the schema, along with fields and resolvers (functions).

Each time a request comes in, GraphQL validates it against the schema, then executes it. It will only execute the query if it refers to the types and fields defined in the schema. GraphQL requests post data to a single endpoint.

By enabling the visualisation of data as a graph and access to multiple data sources with a single request, GraphQL provides a powerful interface for querying your data.

A GraphQL server, meanwhile, can provide information about its schema via introspection. You simply send an introspection query to the GraphQL server. This is handy if you want to do something such as creating a GraphQL proxy in the Tyk Dashboard, as all you need to do is use an introspection query to fetch the schema from the GraphQL upstream.

Why do we need GraphQL federation?

We need GraphQL federation in order to get the best out of GraphQL. Modern businesses usually have dozens of different backend services. Building a single, monolithic GraphQL service isn't the most efficient way to query them. It quickly becomes overly complicated, and hard to maintain, with lots of dependencies.

GraphQL schema federation works by bringing schemas together for a more unified way to access and query data. It replaces the use of schema stitching by providing a more efficient and scalable alternative. With GraphQL federation, you can still expose you multiple backend services as a single graph for consumers.

So, what is a federated graph? It's one that uses a GraphQL federation gateway, with the gateway using metadata from the subgraphs to undertake the required stitching dynamically. What is a federated gateway? It is a gateway such as Tyk, which can look at the different objects in the subgraphs and deliver data flexibly as a result, without the need for you to spend time defining the stitching in the gateway.

GraphQL federation vs schema stitching

Schema stitching was the previous solution for microservice architecture. As such, let's take a quick look at GraphQL federation vs stitching. Both federation and schema stitching do offer the same functionality on the surface, gathering multiple services into one unified gateway, but the implementation is different.

With GraphQL federation, you tell the gateway where it needs to look for the different objects and what URLs they live at. The subgraphs provide metadata that the gateway uses to automatically stitch everything together. This GraphQL federation example is a low-maintenance approach that gives your team a lot of flexibility.

With schema stitching, you must define the "stitching" in the gateway yourself. Your team now has a separate service that needs to be altered, which limits flexibility. The use case for schema stitching is when your underlying services are not all GraphQL. Schema stitching allows you to create a gateway connected to a REST API, for example, while federation only works with GraphQL.

So when should you use either one? Many will say that federation is the overall winner, as it allows teams to focus on their application without needing to maintain a gateway. But if you have different types of APIs, you have to go with schema stitching.

Is GraphQL good for microservices?

Yes, it most certainly is. By using GraphQL for microservices, you can present your clients with a simple interface for interacting with all of your different microservices. You can also add new microservices and make changes to the backend, all without impacting the client experience.

However, this type of platform approach requires careful coordination between frontend and backend teams. The frontend team can change the request/response structure without the backend team's help, but this makes them responsible for security and management. Yet handing responsibility to the backend team means deep coordination is required for maintaining and managing the schema.

This is where a tool such as Universal Data Graph comes in. It means that frontend and backend teams can each focus on their required tasks in a coordinated and collaborative manner, while an API Product Manager nurtures the GraphQL API.

With this structure in place, you have the perfect combination of GraphQL and microservices, at the same time as enjoying peace of mind around security, policies and all the benefits that come with full lifecycle API management.

Does GraphQL replace API gateways?

Yes. No. Well, sort of.

Ok, so there isn't a simple answer to this. API gateways and GraphQL were created for different purposes and fulfil different functions. However, there is some crossover between the two. This means that, in certain circumstances, GraphQL could replace an API gateway. Whether you would want it to is another matter…

Let's use access to internal APIs as an example. GraphQL could deliver an efficient and scalable way to access your GraphQL APIs and retrieve data company-wide, all without an API gateway. However, it doesn't provide the security, access control and usage metrics that are such a fundamental part of an API gateway. So, yes, it's possible to use GraphQL services to replace an API gateway, but doing so means you'll be missing out on features that you really need.

Remember, too, that a gateway can use both GraphQL and REST endpoints, translating requests between the two to avoid any duplication of code. Again, this delivers functionality that GraphQL alone does not.

A better approach is to implement an API management for GraphQL solution that combines the best of both worlds. Tyk's advanced Universal Data Graph is just the ticket. It enables you to adopt GraphQL while also taking care of security, caching, performance optimisation and insights, bringing full lifecycle API management to your graph.

Tyk GraphQL federation implementation

If it's GraphQL federation implementation time, Tyk can help. You get started by creating subgraphs and supergraphs.

Each subgraph represents a backend service. It defines a distinct GraphQL schema that you can query.

As well as querying each service separately using your subgraphs, you can also federate them into a larger schema – a supergraph – that allows you to query multiple backend services at once.

We have included various subgraph examples in our docs that you can copy and paste, along with a supergraph schema example, which shows how to federate your subgraphs in your Tyk Gateway into a single supergraph.

With Tyk, you can quickly and easily create subgraphs via the Dashboard UI. Doing so is just like configuring any other API in Tyk – you just check 'Federation' and 'Subgraph' before you provide an upstream URL and configure your subgraph.

After configuring as many subgraphs as you need, it's simply a matter of checking 'Federation' and 'Supergraph', selecting your subgraphs and configuring your supergraph just as you would any other API in Tyk. Finally, you can define global headers, which are forwarded to all subgraphs that apply to the specific upstream request.

Visit Tyk docs to start your GraphQL federation, from an overview of federation to the nitty gritty of creating subgraphs and supergraphs, and dealing with entity stubs and extension orphans.

Apollo federation implementation

Tyk and Apollo work well in parallel. Following is a general guide on how to implement federation with Apollo.

Creating a gateway

To start, you need to create the gateway. You can use a boilerplate template project, or you can manually install the required packages by running `npm install apollo-server @apollo/gateway graphql`. Once they're installed, create the gateway with a single file `index.js`.

```
const { ApolloServer } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');

const supergraphSdl = ''; // TODO!

const gateway = new ApolloGateway({
  supergraphSdl
});
```

```
const server = new ApolloServer({
  gateway
});

server.listen().then(({ url }) => {
  console.log(`Gateway ready at ${url}`);
}).catch(err => {console.error(err)});
```

Your gateway is ready, but it doesn't have subgraphs connected. To do that, you need to create what's known as the *supergraph*.

Creating the supergraph schema

As with the gateway, there are different ways to create the supergraph. One recommended method is to use the Rover CLI, which needs a YAML file that details where the subgraphs live. Here's an example:

```
subgraphs:
  astronauts:
    routing_url: https://localhost:4001
    schema:
      subgraph_url: https://localhost:4001
  missions:
    routing_url: https://localhost:4002
    schema:
      subgraph_url: https://localhost:4002
```

Name the above file `supergraph-config.yaml`. Run the following command to generate the supergraph schema and output it to `supergraph.graphql`:

```
$ rover supergraph compose --config ./supergraph-config.yaml
> supergraph.graphql
```

Starting the gateway

Now add the supergraph schema to the `index.js` file. Replace the contents of the file with the code below to pull in the contents of the `supergraph.graphql` file you created:

```
const { ApolloServer } = require('apollo-server');
const { ApolloGateway } = require('@apollo/gateway');
const { readFileSync } = require('fs');

const supergraphSdl =
readFileSync('./supergraph.graphql').toString();

const gateway = new ApolloGateway({
  supergraphSdl
});

const server = new ApolloServer({
  gateway
});

server.listen().then(({ url }) => {
  console.log(`Gateway ready at ${url}`);
}).catch(err => {console.error(err)});
```

As you can see, it's not too complicated to get started with GraphQL federation. You can also visit the Apollo GraphQL docs for more information.

Challenges with using GraphQL federation

The greatest difficulty in using federation is figuring out _when_ to use it. Many teams find it improves their infrastructure and increases performance. But that isn't true in all cases.

For instance, does your frontend use many different endpoints to make a single piece of the application function? If so, then federation may be a good choice for you. If you only use one or two endpoints, then the extra complexity added by having to manage a gateway may not be worth it.

Federation requires extra infrastructure, and every new piece added brings more complexity. Also, you will need to work with your services a bit differently. Adding federation may be simple, but adding it effectively takes more work and time. You need to understand the new paradigm it offers and be sure your specific infrastructure can take advantage of it.

Tips for using federation

One of the things you can do to improve your federation experience is building your schema as you go. Federation allows you to break pieces up and use an agile approach to better integrate it so that it keeps running smoothly.

The best way to do that is to divide your services by concern. Take a look at your monolith or your existing microservices and determine their specific purpose. In the example above, missions and astronauts each have their own concerns.

Supports GraphQL federation natively

Allows the easy translation of existing services and data sources into GraphQL subgraphs and supergraphs.

Read full comparison

Good GraphQL federation examples

We've seen some impressive examples of companies harnessing the power of GraphQL. Early adopters such as GitHub, Shopify, Airbnb, Trulia and Expedia have all shown what can be achieved by adopting a GraphQL mindset.

When it comes to GraphQL federation, the leading example has to be Netflix.

Ho w Netflix scales its API with GraphQL federation

Netflix has shared its GraphQL federation journey in detail so that others can learn from its experience. In short, the business needed to address the problem of an increasingly complex backend while still enabling developers to work with one conceptual API. Rapid growth was resulting in consistency and development velocity challenges, so Netflix turned to GraphQL federation to solve these issues.

The priority was an ecosystem that was "unified but decoupled, curated but fast moving" – precisely the advantages that GraphQL federation can deliver. By using federation, Netflix was able to provide a unified API for consumers. It was also able to provide the flexibility and service isolation that backend developers required.

Conclusion

GraphQL federation offers a lot of benefits to developers. It allows you to combine the simplicity of GraphQL with the flexibility of microservices. Hopefully you understand Apollo federation a bit more now too!

As you use federation, you may find you're creating many more APIs. Tyk is a cloud-native API platform that works with GraphQL, provides scalable API lifecycle management for microservice architecture. Tyk's API gateway may just be the right solution for your business.

More on this topic:

- Create, secure and test APIs

CREATE, SECURE & TEST APIS

Share with your network