StepZen

A NEW APPROACH TO GRAPHQL FEDERATION

Anant Jhingran

GraphQL federation is a critical element of your GraphQL architecture. Federation is about assembling the work of independent teams (with GraphQL APIs that represent their domain(s)) into one, or possibly a few, uber GraphQL APIs.

In this paper, we make a case for GraphQL federation and describe the two approaches for stitching graphs together—one based on queries, the other based on objects. Then we discuss how StepZen builds out the query-based stitching model in a simple declarative way and the advantages it gives from code, performance, and governance perspectives.

We end the paper with considerations and potential next steps for you to evaluate a federation model that's right for your organization.

The Case for GraphQL Federation

In many organizations, APIs are built around organizations or domains. For example, a B2C retail business might have customer APIs, e-commerce APIs, marketing APIs, and so on. These APIs are typically built by semi-independent teams, and usually have some common lightweight structures (like authentication, documentation, etc.), but otherwise are constructed independently. And they might appear in the same portal for the API consumer, but there is nothing that connects them.

Because GraphQL APIs are designed to "stitch" data together, GraphQL represents an opportunity to connect APIs easily—even those built by independent teams (see A New Architecture for APIs in The New Stack).

A GraphQL API can answer a query like this: listing all the products a customer has bought and the date they bought them:

```
{
      customer (email: “john.doe@example.com”) {
            name
            purchased {
                        orderedOn
                        name

                  }
            }
      }
}
```

The above snippet stitches `customer` and `order` data into one response. The API to support this GraphQL query might be built by one team, or it might be an assembly of the work of many teams. For the GraphQL API consumer, it does not matter. They get an endpoint that has the data stitched together. This stitching, at its core, is a merge of JSON objects (with both customer data and order data, coming from the respective backends being JSON objects themselves), resulting in a new JSON object. The end result is a JSON object that has the shape of the query.

With this core capability of GraphQL, a simple federation architecture can be built—it is just a tree of graphs, with each subgraph (or a set of subgraphs) managed by different teams.

Copyright © 2022 StepZen Inc. All Rights Reserved.

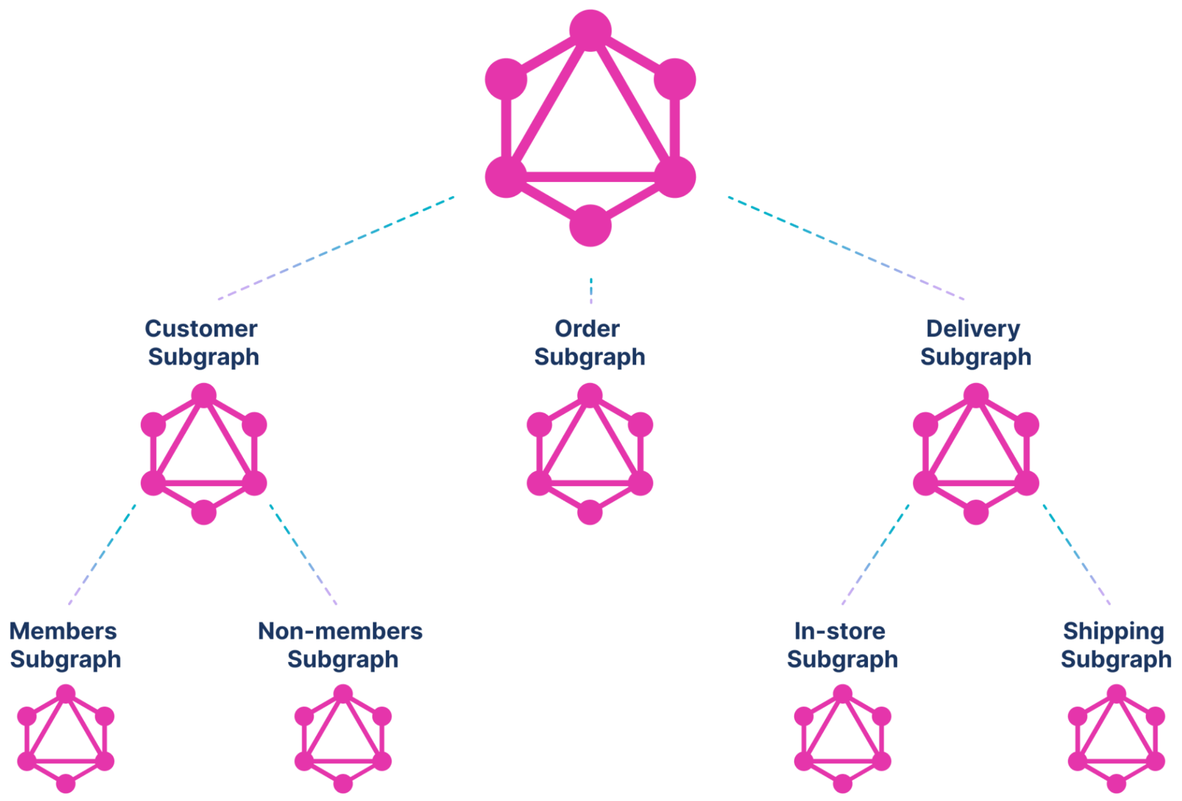


Figure 1. A simple federation architecture: A graph of graphs

You get a set of combined APIs at whatever level you want, and you get independence wherever you want. This is simply not possible in REST APIs without a lot of work. In GraphQL, it is natural.

Copyright © 2022 StepZen Inc. All Rights Reserved.

Different Ways of Approaching Federation

Let us look at the simplest graphs to support the query above:

	Customer Subgraph	**Order Subgraph**

Types	`type Customer {`
	`    id: ID!`
	`    name: String!`
	`    ...`
	`}`
		`type Product {`
		`    id: ID!`
		`    name: String!`
		`    orderedOn: Date!`
		`}`

Queries	`type Query {`
	`   customer(id: ID!):`
	`Customer`
	`   ...`
	`}`
		`type Query {`
		`    productsByCustomer`
		`(customerId: ID!): [Product]`
		`    ...`
		`}`

Table 1. Customer and Order Subgraphs

A federation implementation must be able to present to the user a GraphQL schema that looks like this:

	Federated

Types	`type Customer {`
	`    id: ID!`
	`    name: String!`
	`    `**`purchases:[Product]`**
	`}`
	`type Product {`
	`    id: ID!`
	`    name: String!`
	`    orderedOn: Date!`
	`}`

Queries	`type Query {`
	`   customer(id: ID!): Customer`
	`   purchasesByCustomer (customerId: ID!): [Product]`
	`   ...`
	`}`

Table 2. Federated View of the Graph

The choice of which queries and fields of the two types to expose is up to the federation tier, but for simplicity here, we are exposing all of them in order to focus on the stitching logic. The key question is: how is the execution of the **`Customer.purchases: [Product]`** specified so that the federation tier does the right thing?

Let's examine the two alternatives: query-based stitching and object-based stitching.

Query-based Stitching

StepZen uses query-based stitching. In this model, an edge from `type A` to `type B` is built entirely by `type A`, using a query on `type B`. The syntax is:

```
extend type Customer {
    purchases:[Product]
        @materializer (query: "purchasesByCustomer",
                       arguments: [{name: "customerId", field: "id"}])
}
```

Fundamentally, `Customer.purchases` data is generated by executing the query `purchasesByCustomer` from the `Order` subgraph, passing in the `Customer.id` field as the argument `customerId` in the query. In this world, there is no requirement that the query into the `Order` subgraph must be based on some primary key of the enclosing type. One could instead have (assuming that the `Customer` has a `zipcode`), create a new field like this:

```
extend type Customer {
    popularInNeighborhood:[Product]
        @materializer (query: "popularByZipcode",
                       arguments: [{name: "zipcode", field:
"zipcode"}])
}
```

Both of these queries (`purchasesByCustomer` and `popularByZipcode`) are completely unaware of who might use them. Of course, `type Customer` uses them, but we could have a `type Stock` in the `SupplyChain` subgraph that could also use them. **The responsibility for stitching lies with the enclosing type, not the enclosed type.**

Furthermore, all access controls mechanisms that apply to queries/mutations are available here. Performance optimization is also much simpler. Take caching, for example. Two customers in the same zip code will have the same value for popularInNeighborhood. A query-level cache solves the caching problem.

Object-based Stitching

Apollo uses object-based stitching. In this model, fields of `type A` are populated from multiple subgraphs. Given the federated schema in Table 2, a `Customer` object will get its `name` from the `Customer` subgraph, and `purchases` from the `Order` subgraph. However, unless the two sides agree on some unique identifier, how can the right data be fetched and the correct data be stitched?

That is why object-based stitching requires some unique identifiers. So one might say

`Customer` Subgraph
```
type Customer @key (fields: “id”) {
      id: ID!
      name: String!
}
```

`Order` Subgraph
```
type Customer @key (fields: “id”) {
      id: ID!
      purchased: [Product]
}
```

Now, data from the two subgraphs can be stitched together, but with the following conditions:

- Now, the `Order` subgraph must know about the type it is stitching into. (In contrast, in query-based stitching, it is entirely unaware of its use.) Furthermore, it must not produce any other field that conflicts with the `Customer` subgraph except for the `id` field. And it must name types etc. all the same. Consequently, there must be significant coordination between the teams building the two subgraphs.

  Because there are specific cross dependencies between the two sides, implementations like Apollo add extra directives like `@external`, `@provides`, `@shareable,` etc. This allows tools and federation code to be smarter about reasoning but imposes an extra burden on the teams building the subgraphs. Apollo has decent libraries to manage these, but if teams are not using Apollo, or doing it in different languages, then this becomes especially burdensome.

- Each side must also implement specially named resolvers that take the "`id`" value and return data. The reason for special naming is that the federation code can call the right queries on either side.

- Caching of results is the responsibility of the subgraphs. If we look at the zipcode example above, two customers in the same `zip code` have the same data for `popularInNeghborhood`; however, because it will be generated by a special `id` based resolver, the caching cannot be done at the federation tier. It must be implemented in the subgraphs.

Further discussion of object-based stitching or its Apollo implementation is beyond the scope of this paper. We invite you to do your own comparisons with StepZen's implementation of query-based stitching.

Copyright © 2022 StepZen Inc. All Rights Reserved.

StepZen's Implementation of Query-based Stitching

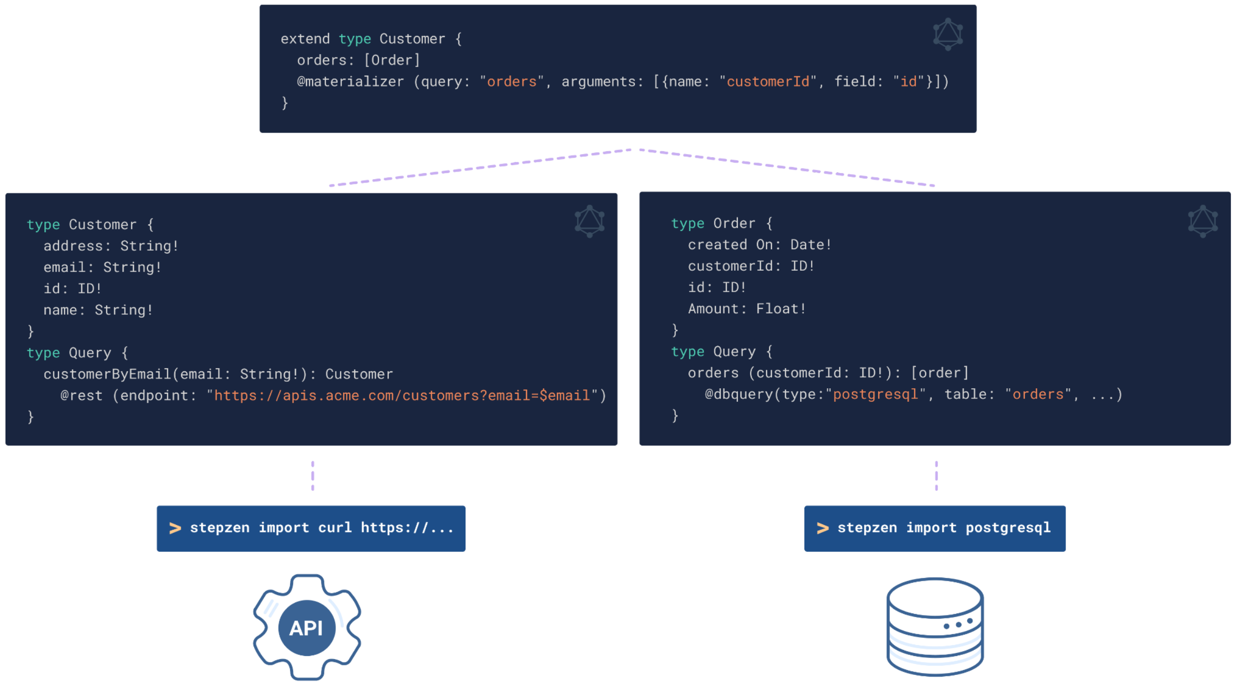
Having built database query engines in the past, we at StepZen are taking our best learnings from the database world and building a new query engine for GraphQL.

StepZen takes a declarative approach to building graphs. A subgraph is formed by connecting the appropriate data source using one of StepZen's GraphQL declarative constructs (GraphQL directives). You can:

- Connect to a REST/SOAP/OData backend using `@rest`
- Connect to a SQL/NoSQL backend using `@dbquery`
- Connect to a GraphQL backend using `@graphql` (and this is what enables easy federation).

Then, a graph of graphs is formed by connecting the data in one graph with a query/mutation in another using `@materializer`. This is the equivalent of `href` in this world. Because `@materializer` does not care how each subgraph that it connects has been built, the structure of a federated graph is the same as the structure of a subgraph.

For example, a team might build a graph like this:



```
extend type Customer {
  orders: [Order]
  @materializer (query: "orders", arguments: [{name: "customerId", field: "id"}])
}
```

```
type Customer {
  address: String!
  email: String!
  id: ID!
  name: String!
}
type Query {
  customerByEmail(email: String!): Customer
    @rest (endpoint: "https://apis.acme.com/customers?email=$email")
}
```

```
type Order {
  created On: Date!
  customerId: ID!
  id: ID!
  Amount: Float!
}
type Query {
  orders (customerId: ID!): [order]
    @dbquery(type:"postgresql", table: "orders", ...)
}
```

```
> stepzen import curl https://...
```

```
> stepzen import postgresql
```

Figure 2. Two subgraphs + one supergraph built using @dbquery, @rest, and @materializer

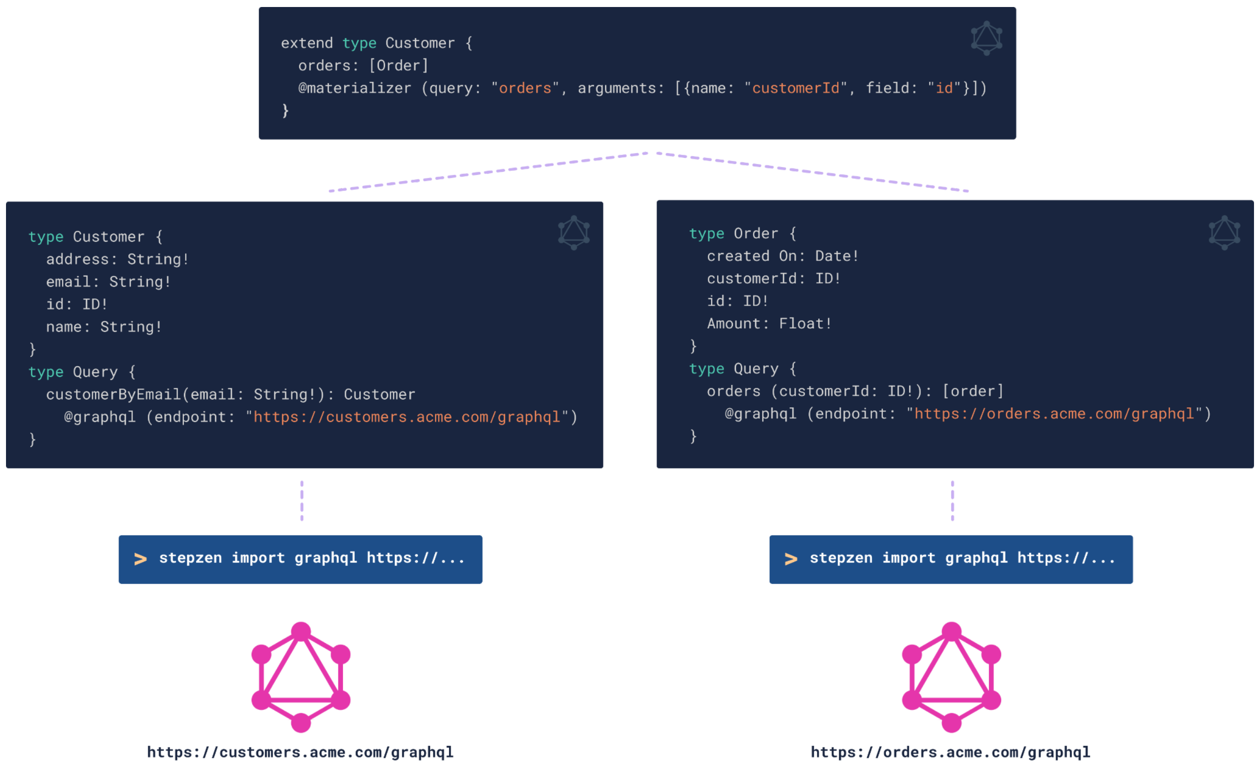
Subgraph1 represents the `customer` domain (this graph is easily generated using `stepzen import curl`).

Subgraph2 represents the `order` domain (this graph is easily generated using `stepzen import postgresql`).

And the graph of graphs is assembled by connecting subgraph2 to subgraph1 — passing data from `Customer` to `Order`, and taking the returned JSON and stitching it into the `Customer` object in a new field called `orders`. Simple, right?

The same approach leads to a simple yet powerful GraphQL federation model.

Assume that instead of one team that is building out both the customer and order domains, there are two teams that have each built out their domain. The technology they use does not matter—they could build it in StepZen, like above, or choose Apollo, Hasura, Kotlin, or another.



```
extend type Customer {
  orders: [Order]
  @materializer (query: "orders", arguments: [{name: "customerId", field: "id"}])
}
```

```
type Customer {
  address: String!
  email: String!
  id: ID!
  name: String!
}
type Query {
  customerByEmail(email: String!): Customer
    @graphql (endpoint: "https://customers.acme.com/graphql")
}
```

```
type Order {
  created On: Date!
  customerId: ID!
  id: ID!
  Amount: Float!
}
type Query {
  orders (customerId: ID!): [order]
    @graphql (endpoint: "https://orders.acme.com/graphql")
}
```

```
> stepzen import graphql https://...
```

```
> stepzen import graphql https://...
```

https://customers.acme.com/graphql
https://orders.acme.com/graphql

Figure 3. Graph of graphs to build federation

Here, Subgraph1, representing the `customer` domain, is available at https://customers.acme.com/graphql.

Also, Subgraph2, representing the `order` domain, is available at https://orders.acme.com/graphql.

The graph of graphs is built by creating two proxies on the `customer` and `order` subgraphs. With StepZen, this is accomplished with one command—`stepzen import graphql`. These proxies fetch data from the corresponding subgraphs using `@graphql`.

And for the federation layer, they look like subgraphs that are local to the construction of the graph of graphs; hence the same `@materializer` can be used to stitch things together.

Furthermore, because the graphs are built and stitched declaratively, StepZen can do optimizations like 1+N, caching, pushdowns, etc. And this is recursively true if the subgraphs are built with StepZen. A declarative approach leads to simpler code and much better runtime characteristics.

A declarative approach leads to simpler code, and better runtime characteristics with built-in optimizations like 1+N, caching, and pushdowns.

In addition to the technical issues discussed above, StepZen's federation model has the following advantages:

Evolution

Going from one team to a "team of teams" is trivial. You keep the `@materializer` code aside, split the rest of the code, create two subgraphs out of it, proxy the subgraphs using `stepzen import graphql`, and throw in your stitching code back in the federation layer. Nothing changes. See our website for more information on how to facilitate teams with federation.

Independence of Concerns

Each subgraph GraphQL service is blissfully unaware of the other. The orders subgraph does not know that it is being federated into the customer subgraph. The only thing that subgraph should care about is what API it exposes further up the chain. How that API is used to federate is not its concern. Keeps the n-square problem from happening at all.

Performance

StepZen's declarative approach allows us to analyze and optimize the execution of the subqueries. We can batch the queries as needed for subgraphs (we detect whether these subqueries take singletons or a batch), and we can and do insert caching at various layers.

Governance

Each subgraph can have its own naming convention. It can have its own authorization/authentication. Because StepZen has sophisticated `@rest` capabilities and our `@graphql` is a layer on top of `@rest`, we can be flexible in mapping different structures, names, and access controls into the federation layer.

Security

GraphQL implementations (including StepZen's) have some powerful security mechanisms for governing who can call what queries with what parameters. Query-based stitching automatically and easily uses the same mechanisms to protect the edges. In contrast, object-based stitching must implement this at the special `@key` resolver level, requiring entirely new mechanisms, which are typically not easily built.

Considering StepZen for Federation

We have described the StepZen approach to federation and why it delivers concisely coded, performance-optimized, well-governed, and secured federation implementations. As you evaluate what's right for your business, we propose a few ways in which you can consider StepZen for federation.

As an Alternative to Apollo Federation

Set up a StepZen account, run the `stepzen import graphql` command against each of your graphs, and if needed, connect them using `@materializer`. Examine the code complexity, and download this performance tool to verify the performance of your system.

As a Subgraph Provider to Apollo Federation

If you have decided on Apollo federation, use StepZen to create a lot more subgraphs—with `stepzen import curl`, `stepzen import mysql, stepzen import postgresql`, or with a few lines of declarative code, you can turn any API or database into a graph.

And then you can federate it in Apollo. StepZen subgraphs are automatically composable into Apollo—there are no new libraries to add, no new types or enhancements to build.

As a Flexible “Any Layer of the Graph” Technology

StepZen has customers excited about how simple and flexible our graph of graph approach is. While attracted by this translating into a simpler federation, they soon build REST and other enhancements into this layer.

In this way, they future-proof themselves for schema and organizational evolution.

Summary

Larger organizations need a way of managing their API sprawl. GraphQL APIs have a natural federation model built-in. There are two approaches to federation:

Query-based stitching

In query-based stitching (which StepZen uses), the edge from type A to type B is entirely built by type A, using a query on type B. This leads to concisely coded, performance-optimized, well-governed, evolvable, and secured federation implementations.

Object-based stitching

In object-based stitching (which Apollo uses), subgraphs contribute fields to the same type. This leads to more complex code and difficulty with governance, security, and performance optimizations. How subgraphs are built and how they are stitched together are different, resulting in more complex evolutions.

Furthermore, StepZen can be the federation layer, or it can slide underneath your current federation layer, making it easy to declare your way through a naturally evolving API landscape.

Copyright © 2022 StepZen Inc. All Rights Reserved.

About the Author

Anant Jhingran is StepZen's CEO and cofounder. Having spent time as IBM Fellow, CTO of IBM's Information Management Division, CTO of Apigee, and product leader at Google Cloud, Anant's career has been at the forefront of innovation in databases, machine learning, and APIs. He has worked on mission critical implementations with companies that range from startups to the Fortune 100. At StepZen, Anant enjoys bringing these technologies together to help simplify, accelerate and scale a new era of API and data-driven development.

.

About StepZen

A word about the StepZen team. We have delivered API software before (at Apigee and at Google), and have built databases like Db2, so we understand query execution and optimization very well. Our vision is to bring the two worlds together—to bring database techniques to the world of APIs. Our goal is to help developers build better GraphQL faster and to ensure successful implementations. Your GraphQL deploys and runs seamlessly on StepZen with built-in performance and reliability optimizations.

We offer the StepZen product, and we offer free workshops on best practices for designing and implementing your system. Whether you are getting started or well on your journey building graphs or "graphs of graphs," we'd love to connect and discuss your use cases.

Copyright © 2022 StepZen Inc. All Rights Reserved.