DZone

GraphQL as an API Gateway to Microservices

by Everett Griffiths MVB · **Feb. 10, 18 · Microservices Zone · Tutorial**

Learn how modern cloud architectures use of microservices has many advantages and enables developers to deliver business software in a CI/CD way.

GraphQL was released to the public back in 2015, and like an animal raised in captivity, its first steps out in the wild were timid and went largely unnoticed. By now, however, it has garnered some major buzz, and with good reason: it solves some of the trickiest problems that are inherent in standard REST API architecture.

Specifically, GraphQL allows you to evolve your API naturally without versioning, it provides workable documentation, it avoids the problems of over- and under-fetching, and it offers a convenient way to aggregate data from multiple sources with a single request. You can start to appreciate its power and flexibility once you get past its unconventional approach and its deceptive "looks-like-JSON" syntax (this coming from the same people who brought us React and its "looks-like-HTML" syntax).

How can GraphQL be leveraged in an API gateway? It seems like it might be a perfect solution for interacting with multiple microservices, each dedicated to a single resource type. Well, the good news is that you *can* use GraphQL in your API Gateway, and it can live alongside standard REST routes. So in some cases, you can have your cake and eat it too.

GraphQL in a Nutshell

Before we get into the gateway code, let's review the landscape of GraphQL a bit. Unlike REST applications, GraphQL implementations rely on a single endpoint. All GraphQL requests post data (*always* post, never get) to that one endpoint with the query that describes which resources and fields are being requested.

GraphQL distinguishes between read operations as "queries" and write operations as "mutations." To support queries, your GraphQL implementation will define a root query object that enumerates all the resource types available for querying (along with their fields and data types). You can think of this as something like a database schema that defines tables and columns.

To support mutations, your GraphQL implementation will define a root mutation object that enumerates all the available mutations and their properties. The mutations can be thought of as actions, *eg*, `createUser` or `updateOrder`. Even within a single language or framework, there is no standard structure for where these GraphQL components should exist, but these critical elements must exist somewhere in any GraphQL implementation.

Using Both GraphQL and REST Endpoints

Just because our API gateway has a GraphQL endpoint defined doesn't mean we can't have other endpoints too! It is entirely possible to define traditional REST routes in the same application.

Since we want to avoid duplicating code in the gateway, especially the code that makes requests out to the microservices, we have to choose which of our methodologies will be in charge. We could either dissect a GraphQL query and translate it into its corresponding REST requests, or we could translate a REST request into its GraphQL equivalent.

It turns out that the latter is simpler, so that's the trick we are proposing: translate requests made to REST routes into GraphQL. No code that interacts with the microservices needs to be duplicated because the REST routes simply act as a translation layer for GraphQL.

In Code

Now that we have described our plan of attack, let's look at some code! The repository that demonstrates these examples is available on GitHub.

This is a Node.js application using the popular Express framework. You should be able to clone and install the application following the instructions in the README.

Running `yarn run start` from the command line should fire up the application and start listening on port 4000. Point your browser to `http://localhost:4000/graphql` to have a look at the GraphQL endpoint. The righthand side will show you any registered resource types and their fields, so right away you can see how GraphQL offers some workable documentation.

We can run an example query to look up a single user by their ID:

```
{
  users(_id: 3){
    name
  }
}
```

```
The result should be:
```

```
{
  "data": {
    "users": [
      {
        "name": "Tammy"
      }
    ]
```

```
  }
}
```

The code that handles this is inside `src/users.js`, and it revolves around the `GraphQLObjectType` object. That's what gives the response its structure and provides workable documentation.

REST Equivalent

Next, let's look at how we might support a REST endpoint that fetches this same data. A conventional route to look up a single user record would follow the pattern of `/users/:userId`. Take a look inside the `index.js` and see how it registered the route:

```
const users = require('./src/rest/user');
// ...
app.use('/users', users);
```

That's pretty standard Express routing stuff. Let's take a look at the `src/rest/user.js` file to see how it converts the request into GraphQL.

```
const app = require('express');
const router = app.Router();
import rootSchema from '../app';

import {graphql} from 'graphql'

const query = (q, vars) => {
    return graphql(rootSchema, q, null, null, vars)
}

// Transform response to JSON API format
// (if desired)
const transform = (result) => {
    const user = result.data.users[0];

    return {
        data: {
            type: 'user',
            id: user._id,
            attributes: {
                name: user.name
            }
        }
    }
}
```

```
// REST request to get a user
router.get('/:userId', (req, res) => {
    // Convert the request into a GraphQL query string
    query("query{users(_id:" + req.params.userId + "){_id, name}}")
        .then(result => {
            const transformed = transform(result)
            res.send(transformed)
        })
        .catch(err => {
            res.sendStatus(500)
        })
})

module.exports = router;
```

This is really where the magic happens. The registered callback for the route assembles the query string and passes it to GraphQL:

```
query("query{users(_id:" + req.params.userId + "){_id, name}}")
```

The query string maps out a query object in that “JSON-like” syntax — yes, it seems redundant, but there is a root query node there that wraps the part of the query that we used on the interactive GraphQL page.

This approach may remind you of the old days before ORMs where you had to assemble query strings by hand. It may be more appropriate to filter the request parameter before putting it into a query string, but since it gets interpreted by GraphQL, it’s probably safe — GraphQL will simply choke if the string is not valid.

Included in this output is a transformer function that alters the default GraphQL response into JSON API format, but that may or may not be something you wish to keep.

You should be able to see this REST endpoint in action by requesting URLs like http://localhost:4000/users/2 and getting responses like:

```
{
    "data": {
        "type": "user",
        "id": "2",
        "attributes": {
            "name": "João"
        }
    }
}
```

Requesting a Microservice

A more complex example involves actually hitting a microservice with a web request. This has been done by requesting an inspirational quote of the day from http://quotes.rest/qod.json?category=inspire

requesting an inspirational quote of the day from http://quotes.rest/qod.json?category=inspire.

In order to add GraphQL support for this data, we need to do three things:

1. Modify the root query object in `src/app.js` .
2. Define the Quote resource type in `src/quote.js` .
3. Define a service that will fetch the quote data from a remote API in `src/services/quote.js` .

Modify the Root Query

First, we need to add the resource type to the GraphQL root query object inside `src/app.js` :

```
// src/app.js
import {
    GraphQLObjectType,
    GraphQLSchema,
} from 'graphql/type';

import userQuery from './users';
import agendaQuery from './agenda-interface';
import quoteQuery from './quote';

const query = new GraphQLObjectType({
    name: 'RootQueryType',
    fields: {
        users: userQuery,
        agenda: agendaQuery,
        quote: quoteQuery
    },
});

export default new GraphQLSchema({
    query,
});
```

Define the Query Type

The quote query object is defined in `src/quote.js` — this is almost exactly the same structure as the one used for the user query, but it references a service class whose job it will be to interact with the remote microservice.

```
// src/quote.js
import {
    GraphQLObjectType,
    GraphQLNonNull,
    GraphQLString
```

```
    GraphQLString
} from 'graphql/type';

import { getQuote } from './services/quote'

export const QuoteType = new GraphQLObjectType({
    name: 'Quote',
    description: 'Quote of the day from API service',
    fields: () => ({
        id: {
            type: GraphQLString,
            description: 'Quote id',
        },
        quote: {
            type: new GraphQLNonNull(GraphQLString),
            description: 'The text of the quote',
        },
        author: {
            type: GraphQLString,
            description: 'The person to whom the quote is attributed',
        },
        date: {
            type: GraphQLString,
            description: 'Date in YYYY-MM-DD format',
        }
    })
});

export default {
    type: QuoteType,
    resolve: getQuote
}
```

This all depends on the `getQuote()` function, which we'll discuss next.

Define a Service for Retrieving Remote Data

The `getQuote()` function is defined inside `src/services/quote.js`:

```
// src/services/quote.js
/**
 * This is where the app calls the microservice responsible for the "Quote" resource type.
 */
import fetch from 'universal-fetch'

```

```
export const
    getQuote = () => {
        const url = 'http://quotes.rest/qod.json?category=inspire'
        return fetch(url)
            .then(response => {
                return response.json()
            })
            .then(json => {
                return transform(json)
            })
            .catch(err => {
                console.trace(err)
            })
    }

// Transform the raw microservice output into
// fields/types defined by the GraphQL type
const transform = (json) => {
    const
        { contents } = json,
        { quotes } = contents,
        quote = quotes[0]

    return {
        id: quote.id,
        quote: quote.quote,
        author: quote.author,
        date: quote.date
    }
}
```

The `transform` method here converts whatever format is used by the microservice into the format defined by GraphQL for this resource type. If you need to add fields to your response, you'll have to add them to your `QuoteType` object in `src/quote.js`.

Once these parts are complete, you should be able to make queries using GraphQL:

```
{
  quote{
    quote,
    author
  }
}
```

Add REST Support

Add REST Support

As before, a route is registered inside the `index.js` file:

```
const quotes = require('./src/rest/quote');
// ...
app.use('/quote', quotes);
```

The REST request acts as a translator to the GraphQL syntax.

```
// src/rest/quote.js
const app = require('express');
const router = app.Router();
import rootSchema from '../app';
import {graphql} from 'graphql'

const query = (q, vars) => {
    return graphql(rootSchema, q, null, null, vars)
}

// Transform response to JSON API format
const transform = (result) => {
    const quote = result.data.quote;

    return {
        data: {
            type: 'quote',
            id: quote.id,
            attributes: {
                quote: quote.quote,
                author: quote.author,
                date: quote.date
            }
        }
    }
}

// REST request to get a quote
router.get('/', (req, res) => {
    // Convert the request into a GraphQL query string
    query("query{quote{id, quote, author, date}}")
        .then(result => {
            const transformed = transform(result)
            res.send(transformed)
        })
        .catch(err => {
```

```
            res.sendStatus(500)
        })
})

module.exports = router;
```

When complete, the REST endpoint is available at http://localhost:4000/quote.

Note that the quote service has a limit of 10 requests per hour, so it the demonstration can only be used in moderation.

Pros

Now that you've seen how you can have your API Gateway be both a GraphQL implementation and support standard REST routes, the benefits should be clear:

- You can have your API cake and eat it too. GraphQL or REST? Both!
- You can leverage GraphQL's built-in strengths of aggregating data from multiple services.

Cons

Most of the disadvantages to this approach have to do with GraphQL in general. The biggest problem with GraphQL when it comes to filling the role of an API gateway is the fact that it operates on a *single* endpoint.

API gateways often define authorization rules, throttling rates, and caching times differently *for each route*. But since GraphQL uses only one endpoint, it's nearly impossible to define route-specific rules for anything. Consequently, you may need to write authorization, throttling, and caching logic in a separate layer or perhaps even in your microservices themselves.

This can create its own mess of smelly code because the solutions will end up undercutting some of the most basic functionality that we expect from the gateway.

If your API is not publicly consumed, then you are not fielding an infinite number of query variations, so having a GraphQL interpreter that allows a client to ask for any possible resource and field combination is arguably overkill. Supporting a handful of use cases with known response attributes has worked quite well for a lot of setups for a long time, so there may be no need to reinvent that particular wheel.

Another disadvantage with trying to use both GraphQL and REST lies with the documentation: there is no guarantee that your REST endpoints have any documentation, let alone docs that stay in sync with the GraphQL query objects, so you are probably inviting some inconsistencies and busywork if you choose to support both GraphQL and REST in your API gateway.

Summary

The solution I have presented here may be an interesting distraction for some, or it may be a viable solution depending on your needs. Although it is possible to have a GraphQL and REST APIs coexist in a single application, the more difficult question is whether or not such a combination is practical.

application, the more difficult question is whether or not such a combination is practical.

Like most of the issues surrounding microservices and API gateways, there are no simple right and wrong answers, there are only tradeoffs, and only you can decide which solutions are the best fit for your needs.

Like This Article? Read More From DZone

VETRO Pattern for API Gateways

Intro To Facebook AccountKit And GraphQL

GraphQL, Just Get Out of My Way and Give Me What I Want

Free DZone Refcard
Microservices in Java

Topics: REST , API GATEWAY , GRAPHQL , MICROSERVICES

Microservices Partner Resources

eploy Pre-Built Sample Microservices OR Create Simple Microservices From Scratch.
oudentity
|
odernizing Application Architectures with Microservices and APIs Virtual Summit Series
A Technologies
|
artner Critical Capabilities Report
A Technologies
|
croservices Architecture eBook: Download Your Copy Here
A Technologies
|