Practical Perl Tools

GraphQL Is Pretty Good Anyway

DAVID N. BLANK-EDELMAN

David has over 30 years of experience in the systems administration/DevOps/SRE field in large multiplatform environments and is the author of the O'Reilly Otter book (new book on SRE forthcoming!). He is one of the co-founders of the now global set of SREcon conferences. David is honored to serve on the USENIX Board of Directors where he helps to organize and engineer conferences like LISA and SREcon. dnb@usenix.org

In a past column we had the pleasure of learning about graph databases together. That particular column was a blast to write because it gave me the opportunity to dig into graphs, something I've always found interesting. In the process of researching that article, I ran into GraphQL. "Oh, goody, more graphs!" I thought. Perhaps an SQL-esque language for graphs? The bad news is GraphQL is nothing like these things or the graph databases we talked about. Even though they both have "graph" in their name, I would be hard-pressed to describe how they connect (truth be told, it isn't immediately apparent why GraphQL has "graph" in the name). The good news is GraphQL is interesting in its own right, so today we are going to give it its own column. And in keeping with my need for radical honesty, I just want to point out up front that the majority of this column will be focused on GraphQL with the Perl bits largely showing up at the end (and being straightforward-ish).

GraphQL Basics

GraphQL describes itself as "a query language for your API," which is both true and perhaps not as helpful as it could be. The official website continues with:

> GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

But I'm still not sure that helps enough. There are a few parts necessary to understanding what's behind GraphQL. To start, I think of it as being one door down from REST on the client-server interaction hallway. To see what I mean, let's use REST as the exemplar since it has been mentioned countless times in this column.

With REST, the dance goes something like this:

- `GET …/items/shoes` the shoes we have

- `GET …/items/shoe/id` the details for a particular shoe

- `GET …/items/shoe/id/laces` the color laces it can come with

- `GET …/stock/id?laces=brown` the number of those shoes with the brown laces in stock

- `GET …/stock/id?laces=black` the number of the black-laced kind in stock

I'm exaggerating a little bit, but with REST the idea is you make a request, then you follow up that request with additional requests for more specific information. Sometimes you do this a bunch of times. This is great from a data architecture perspective (especially if the URLs are legible). This is less great from a "network is slow and perhaps expensive" perspective: for example, if the client was a mobile phone. That was exactly the use case Facebook had in

mind when it created GraphQL. GraphQL attempts to provide a mechanism for saying, "Here's the data I want" and getting it back in a single interaction.

The second thing GraphQL attempts to do is to allow the client to have a simple, clear understanding of just what data the server holds and what the client can ask for. With REST, there's nothing about the interaction model that prevents the client from asking for a piece of fruit from the shoe store or querying `/those-brown-things-that-go-on-your-feet/` instead of the `/shoes/` endpoint. In that example, the server would likely tell the client to take a leap, but wouldn't it be better if the client already had an understanding of what it and the server could correctly chat about? With GraphQL, there is a schema (kinda like database schemas) that is crystal clear about what data is in play, what form it takes, and how it can be queried.

The GraphQL spec says:

> GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.

That's probably the easiest way to think about it.

Let's Play

To get a handle on how this all works in practice (at least at a very surface level), let's look at some sample GraphQL. To give you examples that will be easy for you to explore in greater depth later, I'm going to use ones that resemble those in the official doc on https://graphql.org.

Here's one of the first pieces of GraphQL in the intro tutorial:

```
{
  hero {
    name
  }
}
```

This says to query the field "name" from the hero object. The reply looks (intentionally) like the query:

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

Note, there's something funky in the docs around this example; more info on that in a moment.

We can add more fields and more objects as desired:

```
{
  hero {
    name
       appearsIn
       friends {
         name
       }
  }
}
```

Did you catch the interesting part? Objects can have both fields and sub-objects (that can have fields). In this case, in addition to asking for a new field, I've also asked for both the name fields in the hero object and the name fields in the friends object in that hero object. That would yield something like:

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
              "appearsIn": [
                      "NEWHOPE",
                      "EMPIRE",
                      "JEDI"
              ],
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

This example also shows that, if desired, objects can hold lists of values fields, not just single strings.

If we want to query for a specific object, we can pass in arguments:

```
{
  hero(episode:EMPIRE) {
    name
  }
}
```

and get just the results we need:

```
{
  "data": {
    "hero": {
      "name": "Luke Skywalker"
    }
  }
}
```

Wait, what? If you are puzzled at this response given the material we've seen before, don't sweat it. I was, too. I could not figure out why the initial "{hero {name} }" didn't yield all of the possible heroes. It took me a bunch of spelunking around in the source for the documentation to find the reason, but when I found it, it yielded an important truth. Let me explain.

The reason why we only saw R2-D2 when there wasn't an "episode" argument was this little piece of code called from the source of the page:

```
/* Allows us to fetch the undisputed hero of
   the Star Wars trilogy, R2-D2.
 */
function getHero(episode) {
  if (episode === 'EMPIRE') {
    // Luke is the hero of Episode V.
    return humanData['1000'];
  }
  // Artoo is the hero otherwise.
  return droidData['2001'];
}
```

GraphQL isn't a database. Remember, "GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions." How those interactions take place are (1) language agnostic and (2) defined by the code you do wire up to it. The code assigned for returning heroes (the GraphQL "resolver" for hero) had its own opinion as to what it should return. This particular lesson took me longer to grok than I would have liked; hopefully, I've saved you a little time.

Want to see both heroes? For that, we would use a syntax (aliases) that allows us to ask for two objects that share the same field name, but with different arguments:

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

The result makes a bit more sense now:

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

There are a number of syntactical sugar extensions to the language including those that make it easier to repeat parts of a query without writing it out repeatedly, ways to pass variables into the language, and ways to change the data (mutate it) instead of just querying. There are also some spiffy introspection capabilities that allow a client to ask the server questions about the schema.

In the interest of brevity, rather than diving into these things (or schema construction itself), I recommend you take a look at the tutorial at https://graphql.github.io/learn/. Instead, let's actually see how we can use GraphQL with Perl.

GraphQL and Perl

The heart of all (present day) support of GraphQL in Perl comes from a port of the reference JavaScript implementation. Quick warning: when you install the GraphQL Perl module, it has a number of dependencies. Make that a large number of dependencies (because the dependencies have dependencies). When I installed it on a fresh Perl distribution, the count was 80. I used "cpanm" (which we've talked about in a past column), so it was only a matter of waiting, but I thought I'd give you fair warning.

For the client-server interaction aspect of GraphQL, the client support is pretty trivial. Your client just needs to be able to spit some GraphQL at the server. It could in theory do some more interesting things like schema validation, but let's leave that for a moment. That is probably just by constructing and sending an HTTP request with the right payload in it like we've done a ton of times before in this column. The harder part is the server-side support. That's where the Perl module mostly comes into play.

In the past we've looked at a few Perl web frameworks with the most emphasis on Mojolicious. We'll use Mojolicious::Lite to handle the server duties for this super quick example as well. The key to using Mojolicious is the plugin module called Mojolicious::Plugin::GraphQL, which is a separate dependency you will need to install. Let's take a look at a piece of sample code from a Rosetta Stone-esque blog post here:

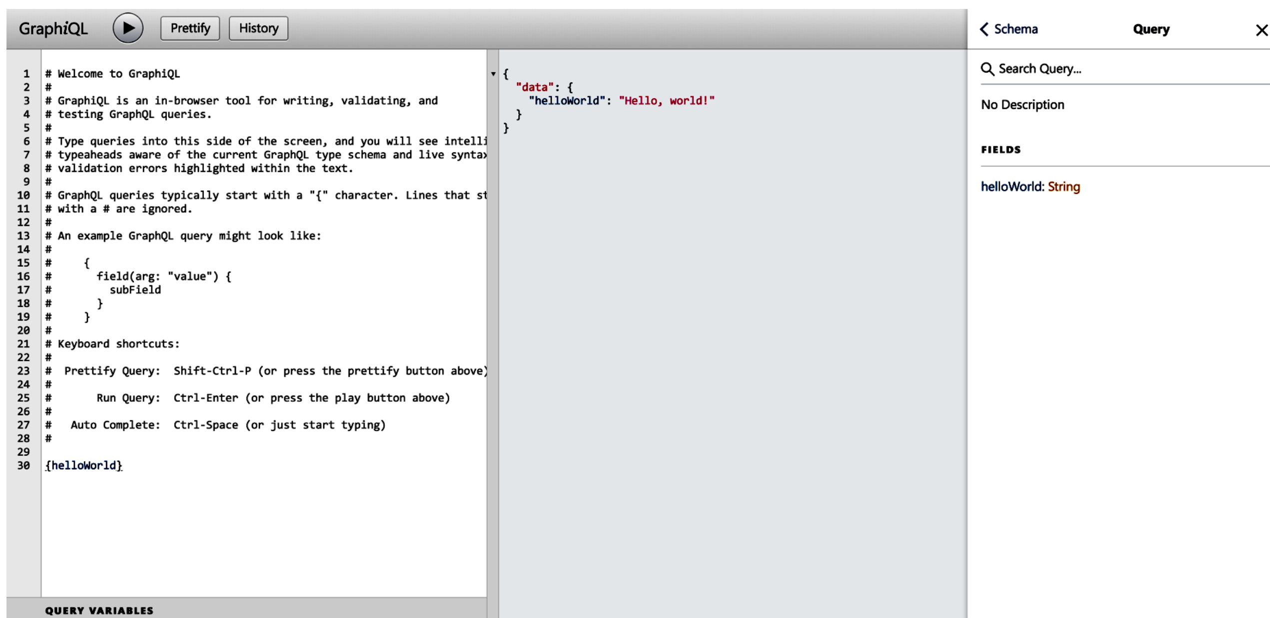

Figure 1: The GraphiQL interface

http://blogs.perl.org/users/ed_j/2017/10/graphql-perl---graphql-js-tutorial-translation-to-graphql-perl-and-mojoliciousplugin-graphql.html.

I call this a Rosetta Stone because this blog post shows the Perl equivalent code for one of the more well-known tutorials whose examples are in JavaScript (https://graphql.org/graphql-js/). Here's one of the code samples from that blog post:

```
use Mojolicious::Lite;
use GraphQL::Schema;
my $schema = GraphQL::Schema->from_doc(<<'EOF');
type Query {
        helloWorld: String
}
EOF
plugin GraphQL => {
        schema => $schema,
        root_value => { helloWorld =>
         'Hello, world!' },
        graphiql => 1,
};
app->start;
```

The first part of the sample includes a definition of a GraphQL schema (a very simple one). The second part loads the GraphQL plugin and sets up the value that will be returned when `{helloWorld}` gets queried. Then we start the Mojolicious event loop and are off to the races.

The one fun part of this plugin shown in the code that I want to highlight is this line:

```
graphiql => 1,
```

GraphiQL is an in-browser IDE that is super spiffy. It allows you to interactively play with GraphQL queries, find errors, see all of the possible objects/fields from a schema, auto-complete them when typing, and so on. When you include this in the plugin configuration as above, it will automatically load GraphiQL for you. So if we start up this code snippet with:

```
$ perl ./test2.pl daemon -l http://*:5000/graphql
[Mon Jun 25 10:43:11 2018] [info] Listening at
"http://*:5000/graphql"
Server available at http://127.0.0.1:5000/graphql
```

and browse to that URL, we see something like Figure 1.

I have opened up the Docs section and clicked through a bit, so you can see that it stands at the ready to show you what's available in the schema. I have also typed something into the left window pane and executed the query, so you can get the full idea from the screen shot.

With this little tip on how to play with GraphQL, I'm going to wind the column down. GraphQL has a bit of a learning curve, but it is some great stuff and there is strong support for it in the community. I hope you'll take a moment to play with it a bit. Take care, and I'll see you next time.