One thing we hear really frequently is how GraphQL makes client-side development more fun, because it can reduce the complexity involved in data loading and management. But in addition to being a great query language, GraphQL is also a specification for how you can use a schema to describe all of the data available in your API, and the relationships between different types in that schema. In this article I'd like to present a new way GraphQL can help you describe your data and make it more easily accessible.

One unified API

GraphQL really shines when you have all of your data in one schema, and you can make a single request to an API to get everything you need. It's already easy to create a GraphQL API that calls into multiple data sources, for example databases or REST APIs. There are tons of examples out there, including the Apollo GitHunt example server which pulls from SQL and a REST API, or this Launchpad example that pulls from multiple rest endpoints in one query.

The next logical question is: What if your microservices, or public APIs that I'm accessing, speak GraphQL? Or what if you want to use a microservice-style deployment model, where your data is split across multiple APIs? To maintain the benefits of GraphQL, we need an easy way to combine all of those GraphQL backends into one schema that can be queried for all of the data at once. And this isn't a new idea — companies like Coursera have already used an approach like this in production for a long time!

Before I go any further, I'd like to credit Mikhail for his awesome research and work on the alpha release of schema stitching we have today, Martijn for helping on the design, Robert for his support of the idea in our conversations, Abhi, Nick, and Tim for helping us understand the requirements, and everyone else who has been part of the conversation for the last few months.

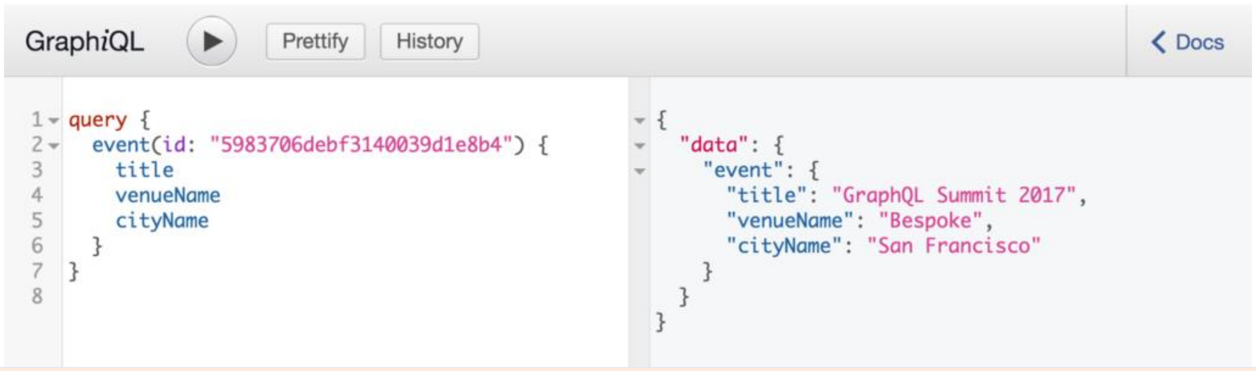
Enter: Schema stitching

Schema stitching is the idea that you can take two or more GraphQL schemas, and merge them into one endpoint that can pull data from all of them. There are a lot of details to figure out, but the general point is easy to grasp: Given some URLs to multiple GraphQL APIs, you want to be able to run a single query that spans across them.

It's easiest to see with an example, so let's look at two related APIs:

1. The new public GraphQL API for Ticketmaster's Universe event management system
2. This Dark Sky weather API on Launchpad, created by Matt Dionis

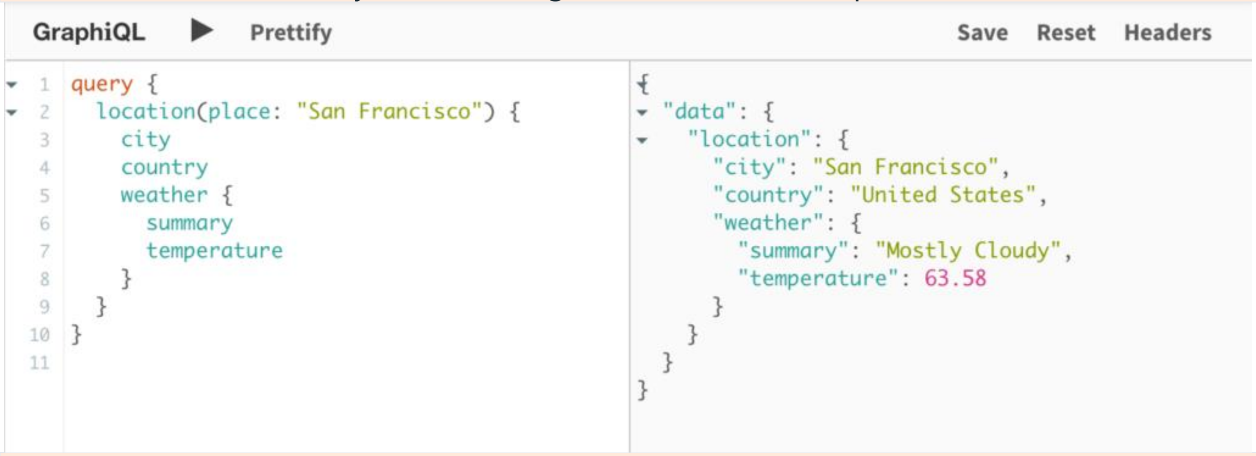
Let's look at two queries we can run against these APIs separately. First, with the Universe API, we can get the details about a specific event ID, in this case GraphQL Summit 2017:



```
query {
  event(id: "5983706debf3140039d1e8b4") {
    title
    venueName
    cityName
  }
}
```

```
{
  "data": {
    "event": {
      "title": "GraphQL Summit 2017",
      "venueName": "Bespoke",
      "cityName": "San Francisco"
    }
  }
}
```

Second, with the Dark Sky API, we can get the weather at a specific location:



```
query {
  location(place: "San Francisco") {
    city
    country
    weather {
      summary
      temperature
    }
  }
}
```

```
{
  "data": {
    "location": {
      "city": "San Francisco",
      "country": "United States",
      "weather": {
        "summary": "Mostly Cloudy",
        "temperature": 63.58
      }
    }
  }
}
```

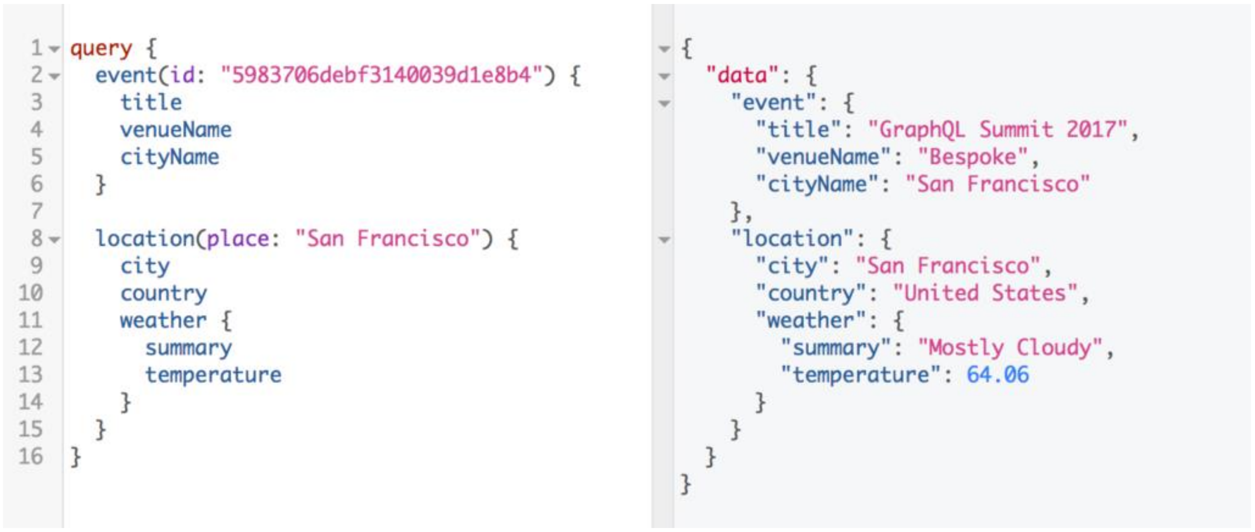
So let’s say we were trying to build an app that could look up a specific event on Universe, and then display the weather at the location of that event. We could definitely do it by running those two queries in our UI, one after the other. But that’s not ideal from a performance and developer experience perspective, since we’ll have to make multiple roundtrips, and keep track of multiple APIs in our frontend code, making sure to send each query to the right one.
So let’s see what it could look like with schema stitching. And by the way, the technology I’m talking about below is currently undergoing active testing, so please join the conversation if you have any feedback!

Simple schema merging

In the simplest implementation of schema stitching, we want to get two benefits:

1. Query against only one schema, so we don’t have to track which queries are associated with which API in our frontend
2. Do one request to get the information we need

So what if we could do an operation to merge the two schemas in such a way that we could just send those two queries side by side? Well, it turns out that we can! Check it out:


```
query {
  event(id: "5983706debf3140039d1e8b4") {
    title
    venueName
    cityName
  }

  location(place: "San Francisco") {
    city
    country
    weather {
      summary
      temperature
    }
  }
}
```

```
{
  "data": {
    "event": {
      "title": "GraphQL Summit 2017",
      "venueName": "Bespoke",
      "cityName": "San Francisco"
    },
    "location": {
      "city": "San Francisco",
      "country": "United States",
      "weather": {
        "summary": "Mostly Cloudy",
        "temperature": 64.06
      }
    }
  }
}
```

Running the two queries as one, using schema stitching. Try this query by downloading and running the example code I created for this post.

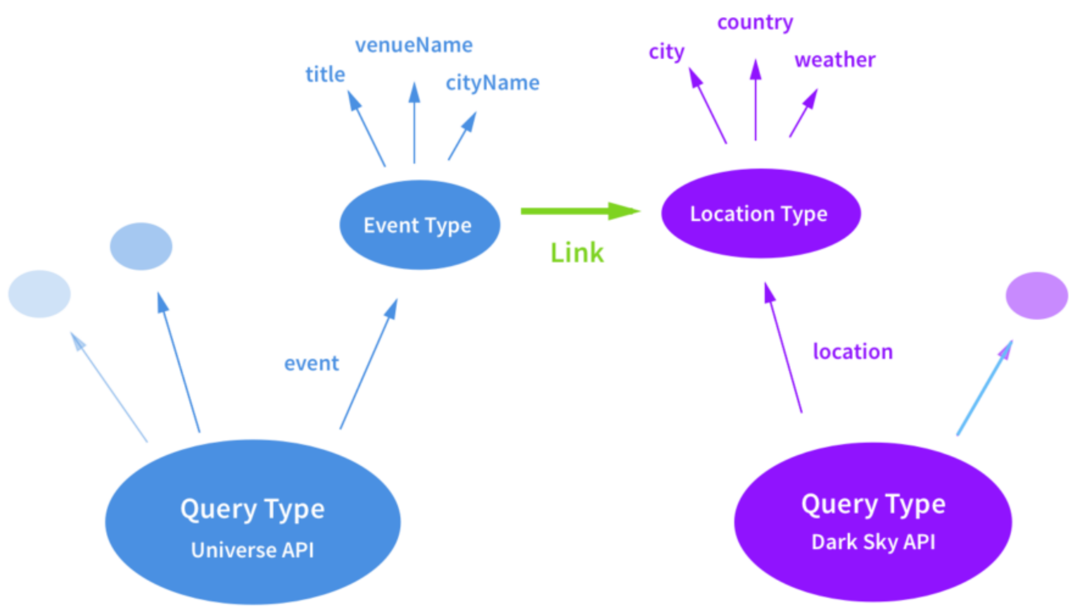
This is super sweet, and it works like this:

1. The server gets the URLs to the two servers, uses GraphQL introspection to discover the schemas, and merges all of the types together.
2. When you run a query, each field on the initial query type (in this case, `event` and `location`) is associated with a particular schema. So the server knows that `event` should go to the Universe API and `location` goes to the Dark Sky API.
3. The server composes queries for those underlying APIs based on the original query fields, fetches them from the backends, and then composes the result back into the shape of the original query.

The proxy server we've created using schema stitching ends up doing exactly two GraphQL requests: One for the first initial field, and one for the second.

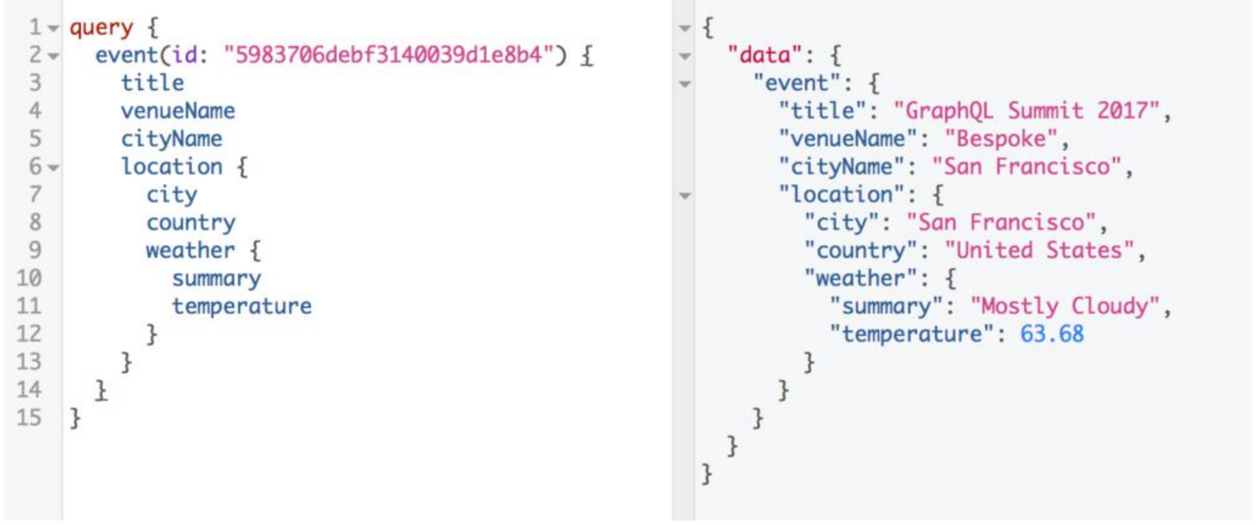
Adding links between types

Ok so the above example isn't exactly what we want, because how would we know that GraphQL Summit is happening in San Francisco if we only have the event ID? For the above server to be useful, the client would have to know all of the arguments up front. But if you look at the first query, we can actually find out the city where the conference is happening from the Universe API. And the Dark Sky API takes the name of the city as an input. Nice! So what if we could pass that `cityName` field as an argument to the `location` field, and not have to write that argument ourselves from the client? This is what we currently call a "link", and it could look like this if you visualized it in a diagram:



A link is a field that bridges two types that originated in different underlying schemas.

Essentially, we want to add a field on the `Event` type from Universe that references the `Location` type from Dark Sky, using the information available on the `Event` type to call the `location` field. And with our current implementation, you can do just that!



```
query {
  event(id: "5983706debf3140039d1e8b4") {
    title
    venueName
    cityName
    location {
      city
      country
      weather {
        summary
        temperature
      }
    }
  }
}
```

```
{
  "data": {
    "event": {
      "title": "GraphQL Summit 2017",
      "venueName": "Bespoke",
      "cityName": "San Francisco",
      "location": {
        "city": "San Francisco",
        "country": "United States",
        "weather": {
          "summary": "Mostly Cloudy",
          "temperature": 63.68
        }
      }
    }
  }
}
```

A query that combines the two above into one roundtrip, and uses the city name to link between the schemas! Download the code here.

Now, we can start to see some more of the power of the schema stitching concept. If we can find common inputs and outputs between schemas (in this case, the name of a location) then we can traverse between related data sets at will. Another example I'd like to build is an API that combines a list of the speakers for GraphQL Summit with their information from the GitHub or Twitter APIs. Unfortunately, Twitter doesn't yet have a public GraphQL API, but we can start with GitHub!

Try the demo

If you'd like to see the code for the examples above, try it for yourself and contribute. Here's how:

1. Download and run the code at stubailo/schema-stitching-demo
2. Follow the PR and post comments at apollographql/graphql-tools#382
3. Test out the alpha and beta releases with your own schemas
4. Join the #schema-stitching channel in the Apollo Community Slack to discuss in real time

We can't wait to hear your ideas and see if you would find this feature useful. It's something I've been personally excited about for a long time, because I think it unveils a lot of new potential for GraphQL to be a language to get all of the data you want.

What's next

I think there's a ton of opportunities here, and we're only scratching the surface with the features above. For me, putting together the demo above felt so natural, like a way that API development should work if we can get the details right. And I think the GraphQL community can figure this out together, especially since similar ideas have been validated at companies like Coursera. Here are some of the use cases that we've been thinking about:

1. Stitching internal schemas to other internal schemas, to ease deployment and versioning
2. Adding public APIs to your schema, to incorporate data from services like GitHub
3. Combining multiple public APIs with some glue to create a mashup API

We've already received and incorporated a lot of feedback from people testing this feature with their own backends, for example:

1. The need for sharing types between the backend schemas
2. Limiting the fields that are available in the public API
3. Keeping the stitched schema in sync with changes in the underlying schemas when they are deployed

So the most important thing we can do is get more information about use cases, and we're looking forward to working with the community on the code as well!
We think that this initial set of functionality is close to a good starting point, so we hope to release something that can be more widely used soon, once we put together some more complete documentation and validate more production use cases. I couldn't be more excited to see what people do with this new way to develop APIs.