/users/get/3500401 you'd write something like

```
{
  user(id: 3500401) {
    id,
    name,
    isViewerFriend,
    profilePicture(size: 50)  {
      uri,
      width,
      height
    }
  }
}
```

which would return something like;

```
{
  "user" : {
    "id": 3500401,
    "name": "Jing Chen",
    "isViewerFriend": true,
    "profilePicture": {
      "uri": "http://someurl.cdn/pic.jpg",
      "width": 50,
      "height": 50
    }
  }
}
```

as opposed to dumping the entire table. How you get that data, process it and use what backends to store it is entirely up to you.

However since GraphQL isn't yet ready for finalization and there are impatient people like me who have jumped the gun and interpreted what this would be by not reading the entire article and tried to infer what it is from the two code snippets that were published.

What my implementation gives you is a graph, an abstract syntax tree to be more specific, what you do with it is entirely up to you.

This isn't a complete implementation of GraphQL seeing at the moment of writing this it wasn't released. If you want to know more about the thing I highly recommend you check out

- GraphQL Introduction
- React.js Conf 2015 - Data fetching for React applications at Facebook

g2sd (Graph to SQL Daemon)

GraphQL to SQL for instance takes in a GraphQL and prints it as SQL for instance

```
{
  Products(ProductID: 3) {
    ProductName,
    UnitsInStock
  }
}
```

and converts it to

```
select ProductName, UnitsInStock from Products where ProductID = 3;
```

which is a better query language because it does look like what we will get back in JSON form, we've also gotten rid of the **select** and

where clauses, but it's not a graph.

Can we represent more complex data structures like this in SQL?

```
{
  Products(ProductID: 3) {
    ProductName,
    UnitsInStock,
    Categories() {
       CategoryName
    }
  }
}
```

for instance this query will be turned in to two queries.

```
select ProductName, UnitsInStock from Products where ProductID = 3;
select CategoryName from Categories ;
```

since there are no params in the second query it will return all the categories in the DB. I'm sure someone will correct me on this because there is a esoteric way this can be achieved I'm not aware off (I'll be happy if you send me a pull request) but at this stage I'm not entirely concerned with writing good SQL queries.

```
{
        "Categories": [
                {
                        "CategoryName": "Beverages"
                },
                {
                        "CategoryName": "Condiments"
                },
                {
                        "CategoryName": "Confections"
                },
                {
                        "CategoryName": "Dairy Products"
                },
                {
                        "CategoryName": "Grains/Cereals"
                },
                {
                        "CategoryName": "Meat/Poultry"
                },
                {
                        "CategoryName": "Produce"
                },
                {
                        "CategoryName": "Seafood"
                }
        ],
        "CategoryID": 2,
        "ProductName": "Aniseed Syrup",
        "UnitsInStock": 13
}
```


Enter variables.

Clearly this is stupid, we won't be needing this all this, we'll be needing the one that carries the same CategoryID as our original query. **GraphQL** allows us to have **Variables** these are marked with

$, in **g2sd** they are able to access their parent result set and use the parents result as a param so this

```
{
  Products(ProductID: 3) {
    ProductName,
    UnitsInStock,
    CategoryId,
    Categories(CategoryID: $CategoryID) {
       CategoryName
    }
  }
}
```

becomes this

```
select ProductName, UnitsInStock, CategoryId from Products where ProductID = 3;
select CategoryName from Categories where CategoryID = 2;
```

and we get

```
{
        "Categories": {
                "CategoryName": "Condiments"
        },
        "CategoryID": 2,
        "ProductName": "Aniseed Syrup",
        "UnitsInStock": 13
}
```

great, you say! You've done something that could have been done with a

join and you did it with two queries and wasted all those cycles. You are a jerk.

First of all dear reader no need for name calling. Secondbly we are not finished.

How about we make this a bit more interesting?

```
{
  Products(ProductID: 11) {
    ProductName,
    UnitsInStock,
    ProductID,
    OrderDetails(ProductID: $ProductID) {
       OrderID,
       Orders(OrderID: $OrderID) {
         EmployeeID,
         Employees(EmployeeID: $EmployeeID) {
            FirstName
         }
       }
    }
  }
}
```

And We'll get something like

```
{
        "OrderDetails": [
                {
                        "OrderID": 10420,
                        "Orders": {
                                "EmployeeID": 3,
                                "Employees": {
                                        "FirstName": "Janet"
                                }
                        }
                },
                ( omitted )
                {
                        "OrderID": 10848,
                        "Orders": {
                                "EmployeeID": 7,
                                "Employees": {
                                        "FirstName": "Robert"
                                }
                        }
                }
        ],
        "ProductID": 9,
        "ProductName": "Mishi Kobe Niku",
        "UnitsInStock": 29
}
```


Here is the SQL that gets us this result.

```
select ProductName, UnitsInStock, ProductID from Products where ProductID = 9;
select OrderID from OrderDetails where ProductID = 9;
select EmployeeID from Orders where OrderID = 10420;
select FirstName from Employees where EmployeeID = 2;
( omitted )
```

Vision for g2sd

So far what we have done is not amazing, and it is certainly not production ready. If you don't see the benefit of writing a declarative queries, how it can speed up your prototyping, or not needing to write any server-side glue code, it's still OK. Because I haven't talked about the vision yet. For that I'm going to borrow, most if not all the features of vitess. I want

- Query rewriting and sanitation Add limits and avoid non-deterministic updates.
- Query blacklisting Customize rules to prevent potentially problematic queries from hitting your database.
- Query killer Terminate queries that take too long to return data.
- Table ACLs Specify access control lists (ACLs) for tables based on the connected user.

In addition to vitess I want to re-write most of the **GraphQL** to **SQL**

bits so they make better use of existing SQL language features. If nothing else comes out of this, I've made a great prototyping tool for my self. And I'm happy with that.

Vision for GraphQL (Go implementation)

I think there is a happy place where GraphQL can live in the SOA world. Things like CloudEndpoints and go-kituse registration mechanisms for handlers, I am trying to figure out the best way to integrate this in to such a work flows. More on that later. b