Introduction

GraphQL Is a Language and a Runtime

The simplest thing we can say about GraphQL is that it's something we put between a front-end application client, and a backend data service.

However, in this simple structure, GraphQL sounds unnecessary, but if you add more clients and more services to the picture, you can start to see the value of a single communication medium. When a client needs access to data from two services together, with GraphQL in the middle, it only needs to communicate with one GraphQL layer and not two different services.

Being a data agent is just one aspect of GraphQL. GraphQL's strong type system and the hierarchal graph-based structure with which GraphQL represents data are two more important aspects and are part of what makes it different.

GraphQL's declarative nature is perhaps the most important aspect about it. GraphQL clients are in control of the data they need. They declare it to the GraphQL servers, and the servers will comply as long as long as the declaration is valid and within their capabilities.

This is different than other APIs where we usually ask about ALL the information available for a single resource. Those APIs make clients over-fetch information that they don't necessarily need.

A standard GraphQL server response would be a JSON object. The answer matches the request query. Every field in the query becomes a key in the answer object, and the key's value is that field's answer. A GraphQL operation determines the shape of its data response, and a data object can be used to easily construct a suitable GraphQL read operation that can be used to ask about that object.

Let's take a quick look at the official GraphQL definition. GraphQL is two parts. It's a language, and it's also a runtime. These two parts are very different.

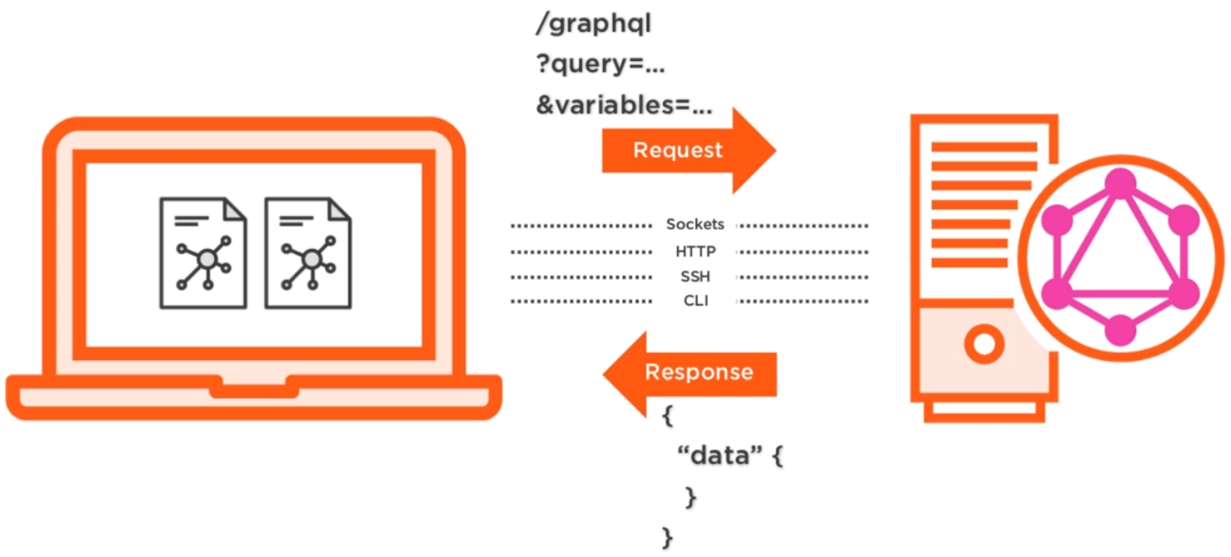
We can use the language to communicate "queries" for reading actions and "mutations" for writing actions. GraphQL also supports real-time data with subscription and a compositional style with fragments.

The runtime has many parts. It has the type system of a GraphQL schema, and it can validate and execute any request. We can also introspectively ask it about its capabilities and supported operations.

GraphQL query Language

A GraphQL operation is just a simple string with optional "variables. " We use variables, for example, in a mutation operation when we want to make it re-usable with different inputs.

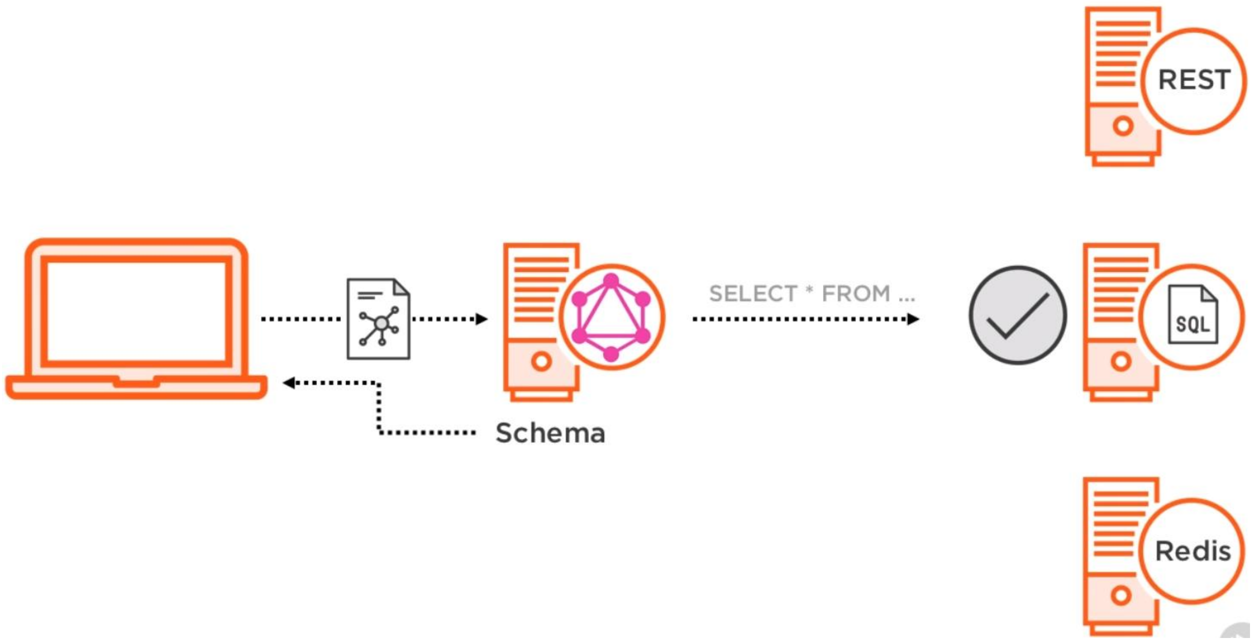
We write GraphQL operations in "documents" on the client-side, and then use an interface to send these documents to a GraphQL server. HTTP is a popular interface option, with which we can send these documents to a GraphQL endpoint like /graphql, with a body data parameter like "query, " and "variables" parameter in case our document uses variables.



The server can also use HTTP to send us back a data response in JSON format for example. HTTP is one option of many. We can, for example, use Sockets, or an SSH protocol, or a simple command-line interface that works with an executable binary. GraphQL does not have any preferences or dependencies on any communication protocol.

Runtime

Just like any other language, a receiver will not understand any communication written in GraphQL without first learning the language or having a translator who understands the language. This is why we need to build a GraphQL runtime as the first layer of our server-side stack. The runtime's job is to understand GraphQL documents and translate them for the other services in the server-side stack. We can add the GraphQL runtime to any existing data service or multiple data services. The data services layers will prepare the data for the request and hand it to the GraphQL engine, which puts it together and sends it to the client who asked for it.



For example, the GraphQL engine can pass a request to a service that fetches information from a REST API or from a SQL server, or from Redis, or a combination of those. The runtime layer can be written in any language, and besides the execution logic that we define there, we also define a graph-based schema to declare our API service capabilities to all clients. Clients can then pick the capabilities they want from the big list we publish.

This approach decouples clients from servers and allows both of them to evolve and scale independently.

Every field we define in a schema has a type. For example, one field can be a scalar value with types like string, integer, or Boolean. Another field might be an object or any array of objects. Every field has a resolver function which can be used to map the field to its read logic.

The GraphQL Editor

GraphQL language is simple and has many similarities to JSON, the JavaScript Object Notation. If you're comfortable working with JSON, you'll be speaking GraphQL in no time.

To learn the language syntax and features in detail we can use GraphQLHub. com for this demo.

When you click on one of these popular APIs on the GraphQLHub home page, you get an in-browser editor, and it's already populated with a sample GraphQL query. This editor is another open source project from Facebook. It's called GraphiQL, and it's an interactive tool that we can use to experiment with and test our GraphQL operations before we use them in our applications. The left side of GraphiQL is where we define the operations, the left bottom corner is where we define any variable values if needed for the operation, and the right side displays the server response when we execute the operation. We can execute this operation with this play button. The server response in this API is a JSON object.

GraphiQL has many features. My favorite of all is the intelligent type-ahead and real-time error highlighting.

But we can also collapse and expand sections of both the operation side and the response side.

We can use this Prettify button if we have a query with bad indentation.

For example, we could copy a query that was generated on a single line to GraphiQL and make it readable using this Prettify button. And, finally, there is an automatically generated documentation about every GraphQL API that we can access using this Docs button.

GraphQL Query Language

Introduction

The GraphQL Query Language is designed around flexible syntax that is easy to read and understand.

Fields – Represents keys which will have some value.

We have two types of fields here--scalar fields and we have complex fields which represents object.

GraphQL Fields (both scalar and complex ones), are modeled after functions. They accept arguments, and they return something in the response. On the server, we'll write actual functions to determine the value returned by every field. We call these functions the resolver functions.

Scalar fields are the basic types in a GraphQL schema. They represent primitive values like strings and integers. In fact, here are all the scalar types we can currently use in a GraphQL schema--integer, float, string, and Boolean.

The fields that represent objects usually have a custom type. For example, the user field has a GithubUser type. This is a type that this particular service defined in its schema, and it's a type that represents an object, and the object has these properties, and every property has its own type.

Spaces, commas, and new lines are all optional in a GraphQL query. And they're used to only make the operation string more readable. They are completely ignored by the GraphQL parser on the server, and in fact, when we hit this Prettify button, anything extra will be removed.

Variables

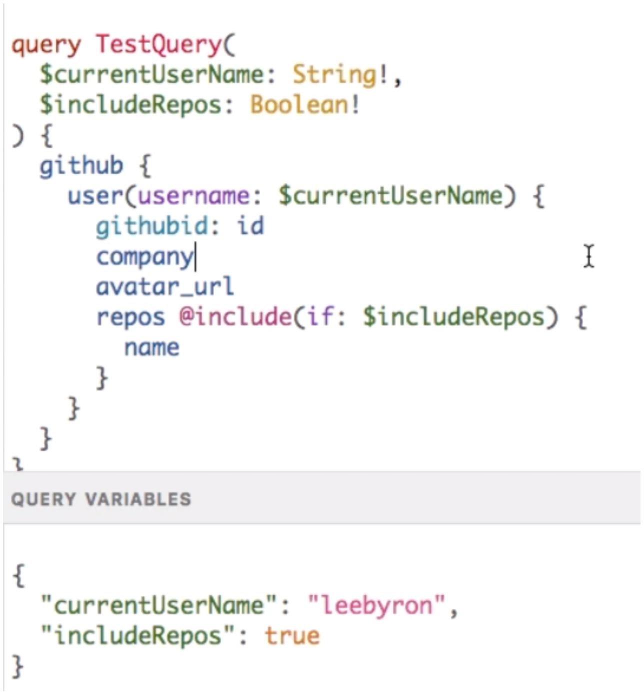
To make it reusable, we need to make it generic by using a GraphQL variable as the input for username. We use a $ sign and a variable name.

Directives

Directives can be used to alter the GraphQL runtime execution, and they are commonly used with variables to customize the response based on the variables values. Examples of built-in directives that should be supported by all GraphQL implementations are the skip and include directives.

A directive is used with the @ sign; it has a name (like include or skip here). That name has to be unique. The directive also accepts a list of arguments, just like fields. For example, the include/skip directives accepts an "if" argument, which is a Boolean.

Aliases



```
query TestQuery(
  $currentUserName: String!,
  $includeRepos: Boolean!
) {
  github {
    user(username: $currentUserName) {
      githubid: id
      company
      avatar_url
      repos @include(if: $includeRepos) {
        name
      }
    }
  }
```

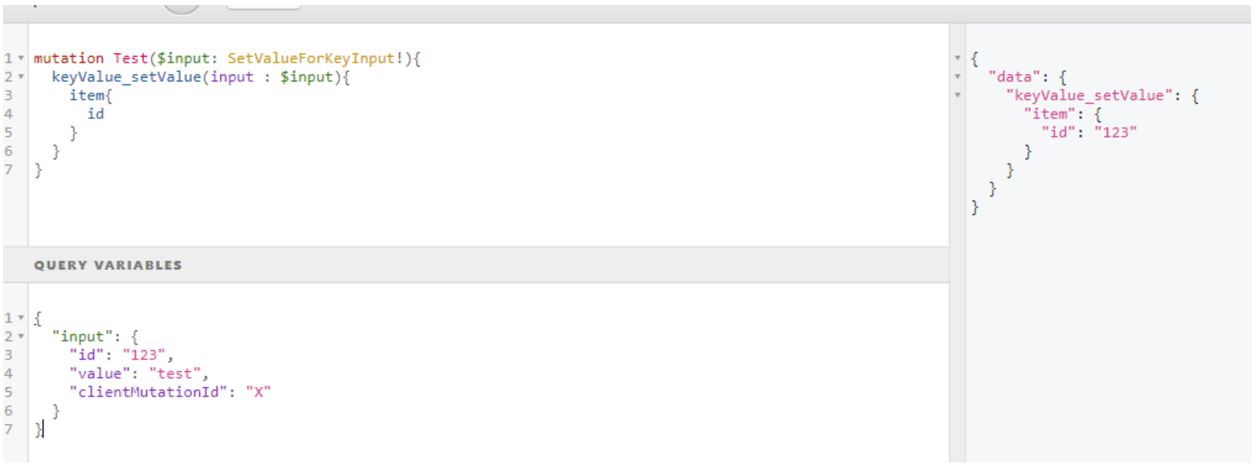
QUERY VARIABLES

```
{
  "currentUserName": "leebyron",
  "includeRepos": true
}
```

Fragments

Fragments let you construct sets of fields, and then include them in queries where you need to.

Mutations



```
mutation Test($input: SetValueForKeyInput!){
  keyValue_setValue(input : $input){
    item{
      id
    }
  }
}
```

QUERY VARIABLES

```
{
  "input": {
    "id": "123",
    "value": "test",
    "clientMutationId": "X"
  }
}
```

```
{
  "data": {
    "keyValue_setValue": {
      "item": {
        "id": "123"
      }
    }
  }
}
```