TECHNICAL WHITE PAPER

An Introduction to GraphQL and Rubrik

Drew Russell
July 2022
RWP-0500

TABLE OF CONTENTS

ABSTRACT

GraphQL is an open source API layer that is utilized by Rubrik CDM and Rubrik Security Cloud. This white paper provides information about consuming the Rubrik GraphQL services, including:

- GraphQL history
- Common GraphQL terms
- The Rubrik GraphQL Playground
- Python, PowerShell, and GoLang usage examples

WHAT IS GRAPHQL?

The GraphQL query language was created to solve the challenges of accessing complex data through RESTful APIs. Instead of having to stitch together multiple REST calls to request data on like objects, GraphQL provides options to create a single query to access the same information. At its heart, GraphQL aims to ease the client/server communication pain points. In other words, ease the communion between your integration or automation and the Rubrik ecosystem.

A GraphQL service uses the HTTP transport method, similar to a REST request.

REQUEST:
Define the query using a JSON like syntax, known as a selection set.

```
query {
  company {
    name
    platform
  }
}
```

RESPONSE:
Only the data requested a response in the same format is provided.

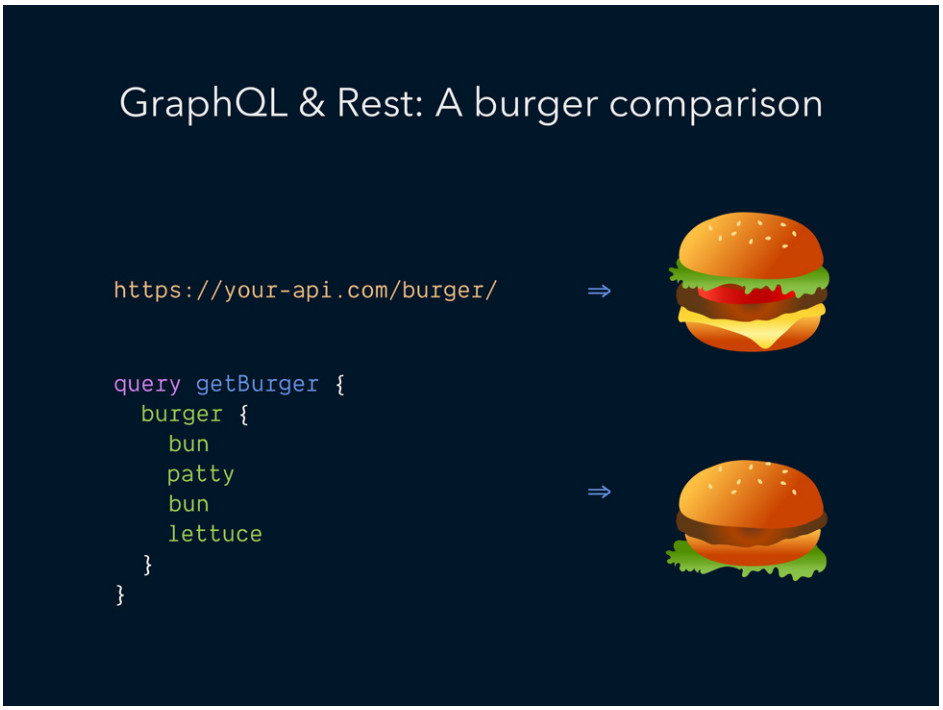
```
{
   "data": {
     "company": {
       "name": "Rubrik",
       "platform": "Cloud Data Management"
     }
   }
 }
```

WHY GRAPHQL?

When fetching information from a REST API, a complete dataset is returned. For example, to request information from two objects, two REST API requests must be sent. The advantage of REST APIs is simplicity — one endpoint does one task, so it's easy to understand and manipulate.

Conversely, when information from a specific endpoint is needed, the request cannot be written to limit the fields that the REST API returns; the response provides a complete data set. This phenomenon is referred to as over fetching. The GraphQL query language provides a language with syntax to tailor the request and return only the needed information, from specific fields within each entity, for multiple objects.

The concept of tailoring your GraphQL request to your needs can be visualized with this burger analogy:



In the case of the REST API, the burger can only be ordered exactly as described on the menu. With GraphQL, instead of getting a burger the way the chef thinks it should be prepared you can request a special order and specify the exact ingredients desired in the order desired.

In essence, GraphQL is extremely powerful, because it provides the option of fetching only the data required, and decreases the amount of processing required is minimized. With automation, the savings really start to add up.

HISTORY OF GRAPHQL

The first draft of GraphQL, known then as SuperGraph, was created at Facebook in February 2012. At the time, the Facebook iPhone app was a simple wrapper for their mobile site that was markedly lacking. So much so that in their shareholder Quarterly Report, Facebook stated that they were "unable to continue to develop products for mobile devices". To address the issues on mobile, Facebook decided to create a new app, from scratch, using native iOS tools. According to Lee Byron, one of the creators of GraphQL, the app "started with our existing APIs and immediately hit issues". Facebook needed a way to better to "request, prepare, and delivery data" to the app.

Enter GraphQL. Working with Lee Byron, Dan Schafer and Nick Schrock were able to design GraphQL in a way that did not rely on individual resources (i.e. individual REST API calls) that would then need to be stitched together but instead would present data, using objects and properties (i.e. a graph), in a more concise and usable format. In August 2012, the first production use of GraphQL was shipped in the new Facebook iPhone app.

After several years of internal use, Lee, Dan, and Nick took a first principle look at GraphQL, applied all lessons learned, and open sourced the first version in July 2015. Within hours, Engineers at Airbnb were diving into the new specification and within a year GitHub released the first public facing GraphQL service.

GRAPHQL AT RUBRIK

Around March 2017 several Rubrik Engineers started a hackathon project to create the first GraphQL service at Rubrik. The initial goal was to stand up a proof of concept to begin exploring the benefits that GraphQL could bring to Rubrik including improving the performance of the Rubrik CDM UI.

Shortly after the initial proof of concept, the Rubrik development team, based off of the results of the hackathon project and a clear indication that the industry was moving more and more towards GraphQL, decided to choose GraphQL as the primary API architecture for the Rubrik Security Cloud.

CORE GRAPHQL CONCEPTS

QUERY

Every GraphQL service has a `query` root type that defines the entry point of the GraphQL query used to fetch data. This is comparable to a `GET` request in a REST API.

QUERY REQUEST

```
query {
  company {
    name
    platform
  }
}
```

QUERY RESPONSE

Only the date requested is provided, in the same format as the query, in the response.

```
{
  cluster(id: "me") {
    version
  }
}
```

MUTATION

The `mutation` root type is used perform an action. This is comparable to a `POST`, `PATCH`, or `PUT` in a REST API.

MUTATION

```
mutation {
  vsphereOnDemandSnapshot(snappableFid: "2918j3k1") {
    id
    status
  }
}
```

FIELD

Fields represent any object defined in the GraphQL service. In the below example, both `cluster` and `version` are fields. Think of fields as the equivalent to keys in a JSON object.

FIELDS

```
{
  cluster(id: "me") {
    version
  }
}
```

NESTED FIELDS

Fields can also be nested. In this example, the `ipmi` and `isAvailable` fields are added to the query.

```
{
  cluster(id: "me") {
    version
    ipmi {
      isAvailable
    }
  }
}
```

ARGUMENTS

Similar to functions in a programming language, each field in a GraphQL query can include arguments to modify the query parameters of the API call.

ARGUMENTS

In this example, `id: "me"` is an argument for the `cluster` field that specifies the specific Rubrik cluster from which to return data.

```
{
  cluster(id: "me") {
    version
    ipmi {
      isAvailable
    }
  }
}
```

ALIASES

The object fields returned by a GraphQL service always match the fields defined in the query, but the returned data does not include the arguments defined in the query.

REQUEST

To determine the operating system of a particular physical host on a Rubrik cluster use the following query:

```
{
  host(id: "Host:::01dcefad") {
    operatingSystem
  }
}
```

RESPONSE

The following data is returned, with the same `host` and `operatingSystem` fields defined in the query:

```
{
  "data": {
    "host": {
      "operatingSystem": "Linux"
    }
  }
}
```

MULTI FIELD REQUEST

What happens when the `host` field is queried multiple times to return data from several hosts?

```
{
  host(id: "Host:::01dcefad") {
    operatingSystem
  }
  host(id: "Host:::02d9332f") {
    operatingSystem
  }
}
```

Because the `host` and `id` arguments are not included in the returned data, there is no way to determine which data corresponds to which query, so GraphQL returns a Field 'host' conflict because they have differing arguments error message.

RENAME FIELD REQUEST

This is where aliases come into play. Aliases can be used to rename the field results. This is shown in the following example with `linuxHost` and `windowsHost` aliases added to the query.

```
{
  linuxHost: host(id: "Host:::01dcefad") {
    operatingSystem
  }
  windowsHost: host(id: "Host:::02d9332f") {
    operatingSystem
  }
}
```

Note: The host field `id` argument example is truncated for readability purposes.

RENAME FIELD RESPONSE

The returned data now includes the defined aliases to further define and organize the response data:

```
{
  "data": {
    "linuxHost": {
      "operatingSystem": "Linux"
    },
    "windowsHost": {
      "operatingSystem": "Windows Server 2016"
    }
  }
}
```

OPERATION NAME

Operation names are an optional, but recommended, construct that adds readability, and thus better maintainability, to code based on the named query.

In the following example the `query` root type alias and the `ClusterDetails` alias, which is the operation name for this query are added.

NAMED OPERATIONS

```
query ClusterDetails {
  cluster(id: "me") {
    version
    ipmi {
      isAvailable
    }
  }
}
```

VARIABLES

Variables are used to replace static argument values with dynamic values. For example a query could utilize logic on a client-side application or script to replace the values in that query but that would unnecessarily add additional overhead. Instead, use the built in variables functionality in GraphQL.

To use variables complete these steps:

1. After the operation name, declare the `$variableName` and `type`. In the following example this corresponds to `$clusterID: String!`.
2. Declare `$variableName` as one of the variables accepted by the query. In the following example this corresponds to `id: $clusterID`.
3. Pass `variableName: value` in a separate, transport-specific (usually JSON) variables dictionary.

OPERATIONS

```
query ClusterDetails($clusterID: String!) {
  cluster(id: $clusterID) {
    version
    ipmi {
      isAvailable
    }
  }
}
```

The variables dictionary passed as JSON.

```
{
  "clusterID": "me"
}
```

FRAGMENTS

Fragments are reusable units used to construct sets of fields and include those sets in queries where needed.

To create a fragment include a name for the fragment (`hostFields`) and specify the field type (`GraphQLHost`) which is defined in the GraphQL service documentation.

Once the fragment has been defined insert it into the query by using `...` + the `framfnent` name (`...hostFields`).

FRAGMENTS

```
fragment hostFields on GraphQlHost {
  operatingSystem
  hostname
  id
  primaryClusterId
}
{
  linuxHost: host(id: "Host:::01c8331f") {
    ...hostFields
  }
  windowsHost: host(id: "Host:::02d9332f") {
    ...hostFields
  }
}
```

Note: The host field `id` argument value is truncated for readability purposes.

CONNECTION

A GraphQL Connection is a standard mechanism for paginating the returned data of a query. to return `X` number of results specific to the query, rather than return an unlimited data response. This mechanism prevents potentially major performance issues.

In the context of Rubrik CDM, a Connection is the mechanism to return `all` data for a particular object type. For example, the `hostConnection` field will look up summary information for all hosts that are registered to a Rubrik cluster. This field is analogous to the `GET /v1/host` REST endpoint.

NODE

In the Rubrik CDM GraphQL service, each `Connection` field contains a `nodes` sub-field that represents the primary data for the object type being queried. For example, the `hostConnection` `nodes` field has various sub-fields such as `hostname`, `operatingSystemType`, and `status`. When a `hostCollection` query is executed, the specified fields are returned for all hosts (i.e. the nodes) on the Rubrik cluster.

The `nodes` field is the main mechanism for retrieving data from the GraphQL service.

```
.
└── hostCollection
    └── nodes
        ├── hostname
        ├── operatingSystemType
        └── status
```

EDGE

The GraphQL specification requires each `Connection` contain an `edges` sub field. In the Rubrik GraphQL service, each `edges` field will contain a `node` and `cursor` sub field. The `cursor` sub field, which allows pagination tracking, is what makes the `edges` field different from calling the `nodes` field directly.

```
.
└── hostCollection
    └── edges
        ├── cursor
        └── node
            ├── hostname
            ├── operatingSystemType
            └── status
```

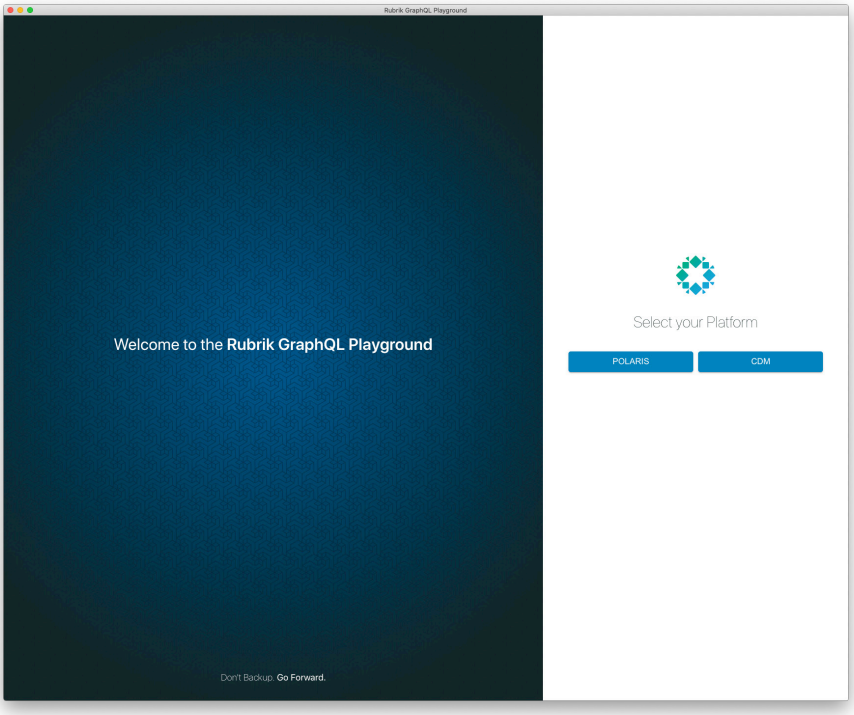
PAGE INFO

The `pageInfo` field, found in each `Connection`, contains the pagination details of the query. The `endCursor` and `hasNextPage` can be used to "turn to the next page of data".

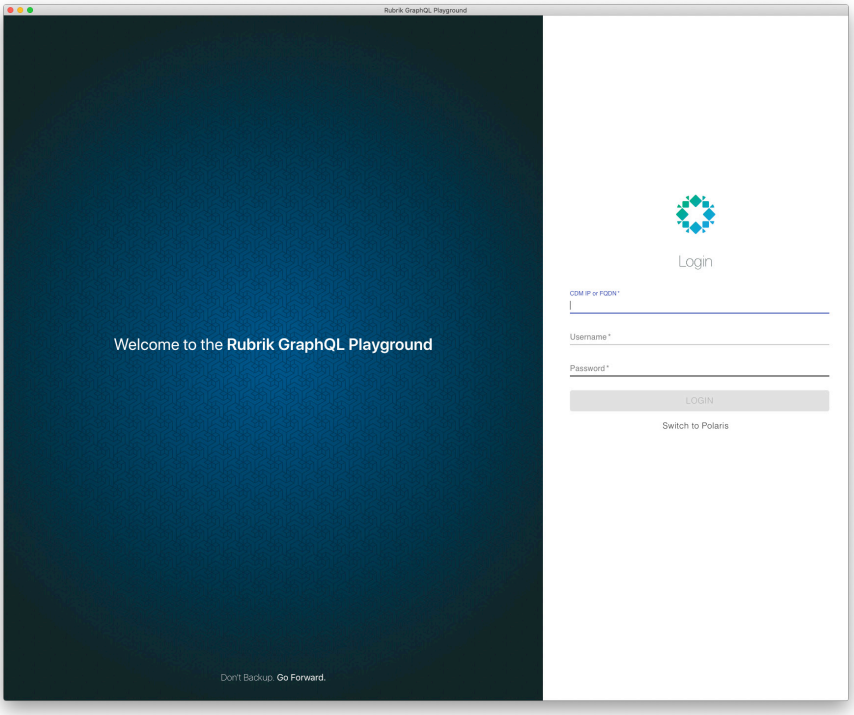
RUBRIK GRAPHQL PLAYGROUND

The Rubrik GraphQL Playground is a cross-platform desktop application based on the open source GraphiQL application. It can be downloaded from the `rubrikinc` GitHub organization: Rubrik GraphQL Playground · GitHub

The application supports both the Rubrik Security Cloud (Polaris) and CDM GraphQL service.

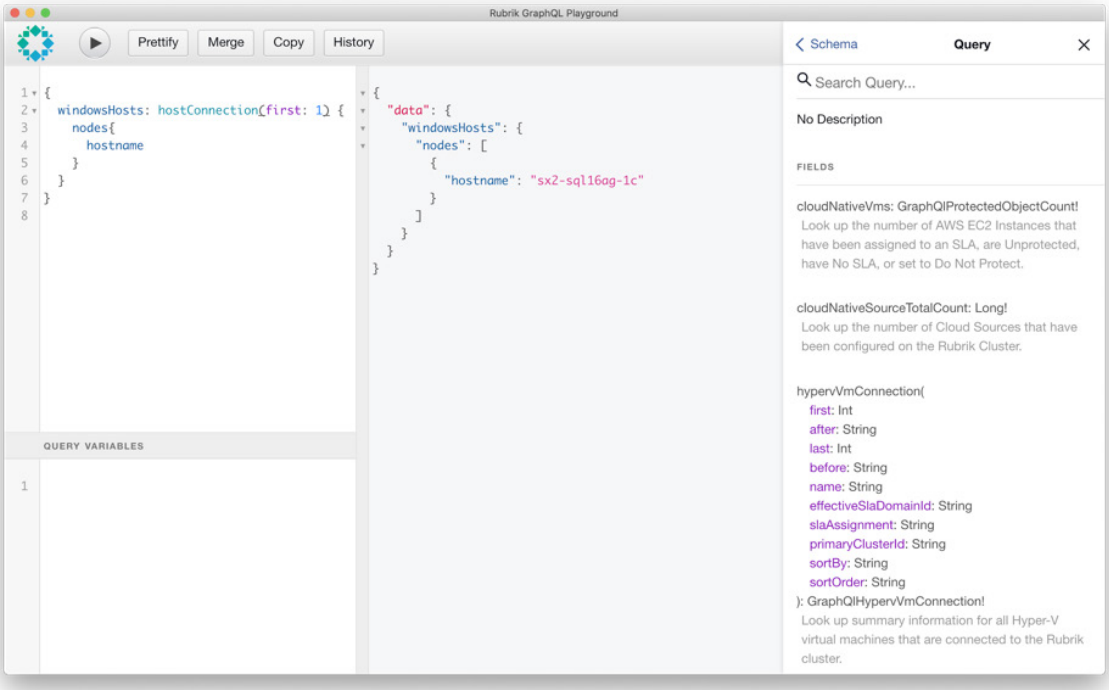


Select the platform and provide the relevant authentication details.



After a successful logon the application opens with three main sections.

- The left section is used to enter the query and optional variables. The query should adhere to standard GraphQL practices and the variables should be formatted as JSON.
- The middle section represents the data returned by the GraphQL service after pressing the "play" button on the top toolbar.
- The right section contains each of the available fields in the service as well as a description of the fields, arguments, and any sub fields.



The top toolbar also includes several helper buttons. The most useful of which are **Prettify**, to format the query text, and **History** to shows the complete query history.

Documentation showing an **!** (exclamation mark) is `non-nullable`. Non-nullable arguments represent a required argument. This means that the GraphQL service always return a value for a query of that field.

CALLING GRAPHQL

Example GraphQL starter scripts in Python, PowerShell, and GoLang.

- **Rubrik Python SDK**
  Rubrik Python SDK script sample
- **Standard Python request**
  Rubrik Python SDK script sample
- **PowerShell**
  PowerShell script sample
- **GoLang**
  GoLang script sample

CONCLUSION

At first glance, GraphQL can be intimidating, especially when compared to REST APIs. Once you begin to learn to basic concepts, everything starts to click and you begin to understand why GraphQL has become so popular in a relatively short period of time. It won't take long until you'll be using a REST endpoint and wish there was a GraphQL alternative. As GraphQL continues to grow in the Rubrik ecosystem our goal is to provide a first-class experience similar to our support of REST endpoints.

ABOUT THE AUTHOR

Drew Russell is a Technical Product Manager focused on the Rubrik API ecosystem. Along with the Rubrik GraphQL and REST API's he has an affinity for Ansible and Python.

VERSION

Version	Date	Summary of Changes
1.0	March 2020	Initial release.
1.1	March 2020	Updated the GraphQL Playground screenshots to reflect changes in the latest version.
1.2	May 2020	Update the GraphQL Playground screenshots to reflect version 2.0 of the application.
1.3	July 2022	Modified Polaris references to reflect that of Rubrik Security Cloud.

SOURCES

- Queries and Mutations | GraphQL
- GraphQL Cursor Connections Specification
- Introduction | Vue Apollo
- GraphQL vs REST: What You Need to Know
- Sara Vieira on Twitter: "GraphQL and Rest Differences explained with burgers 🍑... "
- GraphQL: The Documentary - YouTube
- Facebook Quarterly Report
- GraphQL Landscape

rubrik

Global HQ
3495 Deer Creek Road
Palo Alto, CA 94304
United States

1-844-4RUBRIK
inquiries@rubrik.com
www.rubrik.com

rwp-an-introduction-to-graphql-and-rubrik / 20220725