Chapter 57.

September 2021

Welcome to the September 2021 edition of DataStax Developer's Notebook (DDN). This month we answer the following question(s);

> My company has enjoyed the last two articles on DataStax K8ssandra, and specifically the StarGate component to same. We've seen details on REST and the Document API, but little on GraphQL. Can you help ?
>
> *Excellent question ! We've done a number of articles in this series on GraphQL. Most recently, the October/2020, we delivered a geo-spatial thin client Web program using GraphQL against the DataStax database as a service. When using Astra, the database is hosted, managed. Also when using Astra, the service end points are automatically created and maintained, and are, behind the scenes, using K8ssandra and StarGate.*
>
> *So, in this article, we supply the final and previously missing piece; how to access the GraphQL component of your own hosted K8ssandra/StarGate.*

Software versions

The primary DataStax software component used in this edition of DDN is DataStax Enterprise (DSE), currently release 6.8.*, or DataStax Astra (Apache Cassandra version 4.0.0.*), as required. When running Kubernetes, we are running Kubernetes version 1.17 locally, or on a major cloud provider. All of the steps outlined below can be run on one laptop with 32GB of RAM, or if you prefer, run these steps on Google GCP/GKE, Amazon Web Services (AWS), Microsoft Azure, or similar, to allow yourself a bit more resource

For isolation and (simplicity), we develop and test all systems inside virtual machines using a hypervisor (Oracle Virtual Box, VMWare Fusion version 8.5, or similar). The guest operating system we use is Ubuntu Desktop version 18.04, 64 bit.

57.1 Terms and core concepts

As stated above, ultimately the end goal is to detail GraphQL as it arrives with the DataStax K8ssandra and StarGate component to same. To review the state of the state:

- October/2020, a document in this same series, we delivered a thin client Web program demonstrating geo-spatial queries, using GraphQL queries, against the DataStax Astra hosted Apache Cassandra database service.

  Astra, runs K8ssandra under the hood to automatically provision and maintain service end points using; REST, CQL, GraphQL, and the Document API.
- July/2021, a document in this same series, we detailed installing DataStax K8ssandra; installation, install/verify, basic use with REST.
- August/2021, a document in this same series, continuing from the prior document, we detailed the Document API.
- Recall,
  - GraphQL listens at port 8080
  - REST and the authorization token service listen on port 8081
  - The Document API listens at port 8082
  - So that we need not operate on the same nodes/pods as the hosted Cassandra database server, we can use kubectl port forwarding to gain easy (localhost) access to these ports

So, all that is really left is to detail invoking/using GraphQL as it arrives in a self hosted K8ssandra. The following is assumed moving forward:

- You have a working Kubernetes cluster, with K8ssandra installed to include StarGate.
- You have an Apache Cassandra keyspace, table and data.

Note: The keyspace supporting GraphQL needs to be created via a special means, detailed below.

- You know how to get an 'authorization token' from the Cassandra cluster located within the K8ssandra installation.

Built in Web UI

In an earlier article in this series, we detailed the built in Web UI when working with the Document API to StarGate, at,

```
localhost:8082/swagger-ui
```


Similarly, there is a built in Web UI when working with GraphQL, at,

http://localhost:8080/playground

We've detailed how you must retrieve an authorization key to execute these remote service requests. Additionally, you must set that authorization key value within the Web UI for both the Document API and GraphQL; we've covered this before.

Create a Keyspace

With release 1.0 of DataStax K8ssandra (and we don't know if/when this behavior will change), you must create the keyspace supported StarGate GraphQL queries using the StarGate GraphQL service call as shown below,

```
http://localhost:8080/graphql-schema
      //
   mutation createKsLibrary {
      createKeyspace(name:"ks2", replicas: 1)
   }
```

Relative to the above, the following is offered:

- Above we make a keyspace titled, 'ks2'.
- The Url in our Web browser address bar does not change; we stay on the page hosting this GraphQL Web application.
- The first Url listed above is added as data to the service request on the Web page; this serves as the location of our GraphQL service endpoint.
- We've already discussed setting the authorization token, a header to this service request.
- The last piece is the 'data' of this request, the GraphQL query proper.

  This call is a mutation, the request to create a keyspace. As a keyspace, we run this service request one time only; its results persist.

Create a table

Next we need to make a table, example as shown below.

```
#  Make a tb
   #
   http://localhost:8080/graphql-schema
      //
   mutation createTables {
```



```
    t2: createTable(
      keyspaceName:"ks2",
      tableName:"t2",
      partitionKeys: [
        { name: "col1", type: {basic: TEXT} }
      ]
      clusteringKeys: [
        { name: "col2", type: {basic: TEXT}, order: "ASC" }
          ]
      values: [
        { name: "col3", type: {basic: TEXT} }
        { name: "col4", type: {basic: TEXT} }
      ]
    )
  }
```

Relative to the above, the following is offered:

- The Url to our service request changes, as shown.
- And we call to create a table. In this example; col1 is the partition key, col2 is the clustering column, and columns col3 and col4 are the data proper.

Insert data, get rows

Below is our example to insert two rows in our target tables, the retrieve same. A code review follows.

```
#  Insert rows
   #
   http://localhost:8080/graphql/ks2
      //
   mutation insert_2_recs {
     i1: insertt2(value: {col1:"111", col2:"111", col3:"111",
col4:"111"}) {
       value {
         col1
         col2
```

```
        col3
        col4
      }
    }
    i2: insertt2(value: {col1:"222", col2:"222", col3:"222",
col4:"222"}) {
      value {
        col1
        col2
        col3
        col4
      }
    }
  }

  #  Get rows
  #
  http://localhost:8080/graphql/ks2
     //
  query query_pk {
      t2 (value: {col1:"222"}) {
        values {
          col2
          col3
        }
      }
  }
```

Relative to the above, the following is offered:

- Demonstrating part of the power of GraphQL, the two inserts are written to be submitted via one service invocation.
- And then the SELECT on the partition key.

57.2 Complete the following

Given the past articles in the series as cited, and this new detail, we are now able to run GraphQL queries against a locally hosted K8ssandra, StarGate installation.

We really only covered the basics, a primary key query only; many more advanced query predicates are supported. Further information on reads and writes are available at,

```
https://stargate.io/docs/stargate/1.0/quickstart/quick_start-graphql
.html
```

57.3 In this document, we reviewed or created:

This month and in this document we detailed the following:

- How to run GraphQL queries against a locally hosted K8ssandra, StarGate installation.
- We detailed the need to create a keyspace via given means; currently a requirement when running GraphQL.

Persons who help this month.

Kiyu Gabriel, Jim Hatcher, Joshua Norrid, and Yusuf Abediyeh.

Additional resources:

Free DataStax Enterprise training courses,

```
https://academy.datastax.com/courses/
```


Take any class, any time, for free. If you complete every class on DataStax Academy, you will actually have achieved a pretty good mastery of DataStax Enterprise, Apache Spark, Apache Solr, Apache TinkerPop, and even some programming.

This document is located here,

```
https://github.com/farrell0/DataStax-Developers-Notebook
https://tinyurl.com/ddn3000
```