Quarkus - Using the Cassandra Client

Apache Cassandra® is a free and open-source, distributed, wide column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

In this guide, we will see how you can get your REST services to use a Cassandra database.

ℹ️ This extension is developed by a third party and is part of the Quarkus Platform.

Prerequisites

To complete this guide, you need:

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- GraalVM installed with `GRAALVM_HOME` configured appropriately if you want to use the native mode.

- Apache Maven 3.6.3

- Cassandra or Docker installed

Architecture

The application built in this guide is quite simple: the user can add elements in a list using a form, and the items list is updated.

All the information between the browser and the server is formatted as JSON.

The elements are stored in the Cassandra database.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

The solution is located in the `quickstart` directory.

Creating the Maven project

First, create a new Maven project and copy the `pom.xml` file that is present in the `quickstart` directory.

The `pom.xml` is importing the RESTEasy/JAX-RS, JSON-B, Context Propagation and Cassandra Client extensions.

We will be building a REST application using the [DataStax Object Mapper](#) to simplify the Data Access Layer code.

The most important part of the `pom.xml` is adding the `cassandra-quarkus` extension:

```xml
<dependency>
    <groupId>com.datastax.oss.quarkus</groupId>
    <artifactId>cassandra-quarkus-client</artifactId>
    <version>${quarkus.version}</version>
</dependency>
```

Creating JSON REST service

In this example, we will create an application to manage a list of fruits.

First, let's create the `Fruit` bean as follows:

```java
@Entity
public class Fruit {

    @PartitionKey private String storeId;
    @ClusteringColumn private String name;
    private String description;

    public Fruit() {}

    public Fruit(String storeId, String name, String description) {
      this.storeId = storeId;
      this.name = name;
      this.description = description;
    }

    // getters, setters, hashCode and equals omitted for brevity
}
```

We are using DataStax Java driver Object Mapper, which is why this class is annotated with an `@Entity`. Also, the `storeId` field represents a Cassandra partition key and `name` represents a clustering column, and so we are using the corresponding annotations from the Object Mapper library. It will allow the Mapper to generate proper CQL queries underneath.

> **!** Entity classes are required to have a default no-args constructor.

To leverage the Mapper logic in this app we need to create a DAO:

```
@Dao
public interface FruitDao {
  @Update
  void update(Fruit fruit);

  @Select
  PagingIterable<Fruit> findById(String id);
}
```

This class exposes operations that will be used in the REST service.

Finally, the Mapper itself:

```
@Mapper
public interface FruitMapper {
  @DaoFactory
  FruitDao fruitDao();
}
```

The mapper is responsible for constructing instances of `FruitDao`. In the example above, the `FruitDao` instance will be connected to the same keyspace as the underlying session. More on that below.

> 💡 It is also possible to create DAO instances for different keyspaces. To learn how, see DAO parameterization in the driver docs.

Next, we need a component to create our DAO instances: `FruitDaoProducer`. Indeed, Mapper and Dao instances are stateful objects, and should be created only once, as application-scoped singletons. This component will do exactly that, leveraging Quarkus Dependency Injection container:

```java
import com.datastax.oss.driver.api.core.CqlIdentifier;
import
com.datastax.oss.quarkus.runtime.api.config.CassandraClientConfig;
import
com.datastax.oss.quarkus.runtime.api.session.QuarkusCqlSession;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;
import javax.inject.Inject;

public class FruitDaoProducer {

  private final FruitDao fruitDao;
  private final FruitDaoReactive fruitDaoReactive;

  @Inject
  public FruitDaoProducer(QuarkusCqlSession session) {
    // create a mapper
    FruitMapper mapper = new FruitMapperBuilder(session).build();
    // instantiate our Daos
    fruitDao = mapper.fruitDao();
    fruitDaoReactive = mapper.fruitDaoReactive();
  }

  @Produces
  @ApplicationScoped
  FruitDao produceFruitDao() {
    return fruitDao;
  }

  @Produces
  @ApplicationScoped
  FruitDaoReactive produceFruitDaoReactive() {
    return fruitDaoReactive;
  }
}
```

Note how the `QuarkusCqlSession` instance is injected automatically by the cassandra-quarkus extension in the `FruitDaoProducer` constructor.

Now create a `FruitService` that will be the business layer of our application and store/load the fruits from the Cassandra database.

```
@ApplicationScoped
public class FruitService {

  private final FruitDao dao;

  @Inject
  public FruitService(FruitDao dao) {
    this.dao = dao;
  }

  public void save(Fruit fruit) {
    dao.update(fruit);
  }

  public List<Fruit> get(String id) {
    return dao.findById(id).all();
  }
}
```

Note how the service receives a `FruitDao` instance in the constructor. This DAO instance is provided by `FruitDaoProducer` and injected automatically.

The last missing piece is the REST API that will expose GET and POST methods:

```java
@Path("/fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {

  private static final String STORE_NAME = "acme";

  @Inject FruitService fruitService;

  @GET
  public List<FruitDto> list() {
    return fruitService.get(STORE_NAME).stream()
        .map(fruit -> new FruitDto(fruit.getName(),
fruit.getDescription()))
        .collect(Collectors.toList());
  }

  @POST
  public void add(FruitDto fruit) {
    fruitService.save(covertFromDto(fruit));
  }

  private Fruit covertFromDto(FruitDto fruitDto) {
    return new Fruit(fruitDto.getName(), fruitDto.getDescription(),
STORE_NAME);
  }
}
```

The `list` and `add` operations are executed for the `storeId` "acme". This is the partition key of our data model. We can easily retrieve all rows from cassandra using that partition key. They will be sorted by the clustering column. `FruitResource` is using `FruitService` which encapsulates the data access logic.

When creating the REST API we should not share the same entity object between REST API and data access layers. They should not be coupled to allow the API to evolve independently of the storage layer. This is the reason why the API is using a `FruitDto` class. This class will be used by Quarkus to convert JSON to java objects for client requests and java objects to JSON for the responses. The translation is done by quarkus-resteasy extension.

```
public class FruitDto {

  private String name;
  private String description;

  public FruitDto() {}

  public FruitDto(String name, String description) {
    this.name = name;
    this.description = description;
  }
  // getters and setters omitted for brevity
}
```

> ❗ DTO classes used by the JSON serialization layer are required to have a default no-arg constructor.

Configuring the Cassandra database

Connecting to Apache Cassandra or DataStax Enterprise (DSE)

The main properties to configure are: `contact-points`, to access the Cassandra database, `local-datacenter`, which is required by the driver, and – optionally – the keyspace to bind to.

A sample configuration should look like this:

```
quarkus.cassandra.contact-points={cassandra_ip}:9042
quarkus.cassandra.local-datacenter={dc_name}
quarkus.cassandra.keyspace={keyspace}
```

In this example, we are using a single instance running on localhost, and the keyspace containing our data is `k1`:

```
quarkus.cassandra.contact-points=127.0.0.1:9042
quarkus.cassandra.local-datacenter=datacenter1
quarkus.cassandra.keyspace=k1
```

If your cluster requires plain text authentication, you can also provide two more settings: `username` and `password`.

```
quarkus.cassandra.auth.username=john
quarkus.cassandra.auth.password=s3cr3t
```

Connecting to a cloud DataStax Astra database

When connecting to Astra, instead of providing a contact point and a datacenter, you should provide `secure-connect-bundle`, which should point to a valid path to an Astra secure connect bundle, as well as `username` and`password`, since authentication is always required on Astra clusters.

A sample configuration for DataStax Astra should look like this:

```
quarkus.cassandra.cloud.secure-connect-bundle=/path/to/secure-
connect-bundle.zip
quarkus.cassandra.auth.username=john
quarkus.cassandra.auth.password=s3cr3t
quarkus.cassandra.keyspace=k1
```

Advanced driver configuration

You can configure other Java driver settings using `application.conf` or `application.json` files. They need to be located in the classpath of your application. All settings will be passed automatically to the underlying driver configuration mechanism. Settings defined in `application.properties` with the `quarkus.cassandra` prefix will have priority over settings defined in `application.conf` or `application.json`.

To see the full list of settings, please refer to the driver settings reference.

Running a Cassandra Database

By default, `CassandraClient` is configured to access a local Cassandra database on port 9042 (the default Cassandra port).

> ❗ Make sure that the setting `quarkus.cassandra.local-datacenter` matches the datacenter of your Cassandra cluster.

> 💡 If you don't know the name of your local datacenter, this value can be found by running the following CQL query: `SELECT data_center FROM system.local`.

If you want to use Docker to run a Cassandra database, you can use the following command to launch one:

```
docker run \
    --name local-cassandra-instance \
    -p 7000:7000 \
    -p 7001:7001 \
    -p 7199:7199 \
    -p 9042:9042 \
    -p 9160:9160 \
    -p 9404:9404 \
    -d \
    launcher.gcr.io/google/cassandra3
```

Note that only the 9042 port is required. All others all optional but provide enhanced features like JMX monitoring of the Cassandra instance.

Next you need to create the keyspace and table that will be used by your application. If you are using Docker, run the following commands:

```
docker exec -it local-cassandra-instance cqlsh -e "CREATE KEYSPACE
IF NOT EXISTS k1 WITH replication = {'class':'SimpleStrategy',
'replication_factor':1}"
docker exec -it local-cassandra-instance cqlsh -e "CREATE TABLE IF
NOT EXISTS k1.fruit(id text, name text, description text, PRIMARY
KEY((id), name))"
```

If you're running Cassandra locally you can execute the cqlsh commands directly:

```
cqlsh -e "CREATE KEYSPACE IF NOT EXISTS k1 WITH replication =
{'class':'SimpleStrategy', 'replication_factor':1}
cqlsh -e "CREATE TABLE IF NOT EXISTS k1.fruit(id text, name text,
description text, PRIMARY KEY((id), name))
```

Creating a frontend

Now let's add a simple web page to interact with our `FruitResource`.

Quarkus automatically serves static resources located under the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, add a `fruits.html` file with the content from this fruits.html file in it.

You can now interact with your REST service:

- start Quarkus with `mvn clean quarkus:dev`
- open a browser to `http://localhost:8080/fruits.html`
- add new fruits to the list via the form

Reactive Cassandra Client

When using `QuarkusCqlSession` you have access to reactive variant of methods that integrate with Quarkus and Mutiny.

> If you're not familiar with Mutiny, read the Getting Started with Reactive guide first.

Let's rewrite the previous example using reactive programming with Mutiny.

Firstly, we need to implement the `@Dao` that works in a reactive way:

```
@Dao
public interface FruitDaoReactive {

  @Update
  Uni<Void> updateAsync(Fruit fruitDao);

  @Select
  MutinyMappedReactiveResultSet<Fruit> findByIdAsync(String id);
}
```

Please note the usage of `MutinyMappedReactiveResultSet` - it is a specialized `Mutiny` type converted from the original `Publisher` returned by the driver, which also exposes a few extra methods, e.g. to obtain the query execution info. If you don't need anything in that interface, you can also simply declare your method to return `Multi`: `Multi<Fruit> findByIdAsync(String id)`,

Similarly, the method `updateAsync` returns a `Uni` - it is automatically converted from the original result set returned by the driver.

> The Cassandra driver uses the Reactive Streams `Publisher` API for reactive calls. The Quarkus framework however uses Mutiny. Because of that, the `CqlQuarkusSession` interface transparently converts the `Publisher` instances returned by the driver into the reactive type `Multi`. `CqlQuarkusSession` is also capable of converting a `Publisher` into a `Uni` – in this case, the publisher is expected to emit at most one row, then complete. This is suitable for write queries (they return no rows), or for read queries guaranteed to return one row at most (count queries, for example).

Next, we need to adapt the `FruitMapper` to construct a `FruitDaoReactive` instance:

```
@Mapper
public interface FruitMapper {
  // the existing method omitted

  @DaoFactory
  FruitDaoReactive fruitDaoReactive();
}
```

Now, we can create a `FruitReactiveService` that leverages the reactive `@Dao`:

```
@ApplicationScoped
public class FruitReactiveService {

  private final FruitDaoReactive fruitDao;

  @Inject
  public FruitReactiveService(FruitDaoReactive fruitDao) {
    this.fruitDao = fruitDao;
  }

  public Uni<Void> add(Fruit fruit) {
    return fruitDao.update(fruit);
  }

  public Multi<Fruit> get(String id) {
    return fruitDao.findById(id);
  }
}
```

> The `get()` method above returns `Multi`, and the `add()` method returns `Uni`; these types are compatible with the Quarkus reactive REST API.

To integrate the reactive logic with REST API, you need to have a dependency to `quarkus-resteasy-mutiny`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-mutiny</artifactId>
</dependency>
```

It provides an integration layer between `Multi`, `Uni` and the REST API.

Finally, we can create a `FruitReactiveResource`:

```

```
@Path("/reactive-fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitReactiveResource {
  private static final String STORE_NAME = "acme";
  @Inject FruitReactiveService service;

  @GET
  public Multi<FruitDto> getAll() {
    return service
        .get(STORE_NAME)
        .map(fruit -> new FruitDto(fruit.getName(),
fruit.getDescription())));
  }

  @POST
  public Multi<FruitDto> add(FruitDto fruitDto) {
    Fruit fruit = covertFromDto(fruitDto);
    return service.add(fruit).then(ignored -> getAll());
  }

  private Fruit covertFromDto(FruitDto fruitDto) {
    return new Fruit(fruitDto.getName(), fruitDto.getDescription(),
STORE_NAME);
  }
}
```

> ℹ️ All methods exposed via REST interface are returning reactive types from the Mutiny API.

# Creating a reactive frontend

Now let's add a simple web page to interact with our `FruitReactiveResource`. In the `src/main/resources/META-INF/resources` directory, add a `reactive-fruits.html` file with the content from this reactive-fruits.html file in it.

You can now interact with your reactive REST service:

- start Quarkus with `mvn clean quarkus:dev`

- open a browser to `http://localhost:8080/reactive-fruits.html`

- add new fruits to the list via the form

# Connection Health Check

If you are using the `quarkus-smallrye-health` extension, `cassandra-quarkus` will

automatically add a readiness health check to validate the connection to the cluster.

So when you access the `/health/ready` endpoint of your application you will have information about the connection validation status.

> 💡 This behavior can be disabled by setting the `quarkus.cassandra.health.enabled` property to `false` in your `application.properties`.

# Metrics

If you are using the `quarkus-smallrye-metrics` extension, `cassandra-quarkus` can provide metrics about QuarkusCqlSession and Cassandra nodes.

> 💡 This behavior must first be enabled by setting the `quarkus.cassandra.metrics.enabled` property to `true` in your `application.properties`.

The next step that you need to do is set explicitly which metrics should be enabled.

The `quarkus.cassandra.metrics.session-enabled` and `quarkus.cassandra.metrics.node-enabled` properties should be used for enabling metrics; the former should contain a list of session-level metrics to enable, while the latter should contain a list of node-level metrics to enable. Both properties accept a comma-separated list of valid metric names.

For example, to enable `session.connected-nodes`, `session.bytes-sent`, and `node.pool.open-connections` you should add the following settings to your `application.properties`:

```
quarkus.cassandra.metrics.enabled=true
quarkus.cassandra.metrics.session-enabled=connected-nodes,bytes-
sent
quarkus.cassandra.metrics.node-enabled=pool.open-connections
```

For the full list of available metrics, please refer to the driver settings reference and the `advanced.metrics` section.

When metrics are properly enabled and when you access the `/metrics` endpoint of your application, you will see metric reports for all enabled metrics.

Building a native executable

You can use the Cassandra client in a native executable.

You can build a native executable with the `mvn clean package -Dnative` command.

Running it is as simple as executing `./target/quickstart-1.0.0-SNAPSHOT-runner`.

You can then point your browser to `http://localhost:8080/fruits.html` and use your application.

Conclusion

Accessing a Cassandra database from a client application is easy with Quarkus and the Cassandra extension, which provides configuration and native support for the DataStax Java driver for Apache Cassandra.