Quarkus - Neo4j

Neo4j is a graph database management system developed by Neo4j, Inc. Neo4j is a native graph database focused not only on the data itself, but especially on the relations between data. Neo4j stores data as a property graph, which consists of vertices or nodes as we call them, connected with edges or relationships. Both of them can have properties.

Neo4j offers Cypher, a declarative query language much like SQL. Cypher is used to for both querying the graph and creating or updating nodes and relationships. As a declarative language it used to tell the database what to do and not how to do it.

> ℹ️ Learn more about Cypher in the Neo4j Cypher manual. Cypher is not only available in Neo4j, but for example coming to Apache Spark. A spec called OpenCypher is available, too.

Clients communicate over the so called Bolt protocol with the database.

Neo4j - as the most popular graph database according to DB-Engines ranking - provides a variety of drivers for various languages.

The Quarkus Neo4j extension is based on the official Neo4j Java Driver. The extension provides an instance of the driver configured ready for usage in any Quarkus application. You will be able to issue arbitrary Cypher statements over Bolt with this extension. Those statements can be simple CRUD statements as well as complex queries, calling graph algorithms and more.

The driver itself is released under the Apache 2.0 license, while Neo4j itself is available in a GPL3-licensed open-source "community edition", with online backup and high availability extensions licensed under a closed-source commercial license.

> ℹ️ This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

Programming model

The driver and thus the Quarkus extension support three different programming models:

- Blocking database access (much like standard JDBC)

- Asynchronous programming based on JDK's completable futures and related infrastructure

- Reactive programming based on Reactive Streams

The reactive programming model is only available when connected against a 4.0+ version of Neo4j. Reactive programming in Neo4j is fully end-to-end reactive and therefore requires a server that supports backpressure.

In this guide you will learn how to

- Add the Neo4j extension to your project

- Configure the driver

- And how to use the driver to access a Neo4j database

This guide will focus on asynchronous access to Neo4j, as this is ready to use for everyone. At the end of this guide, there will be a reactive version, which needs however a 4.0 database version.

The domain

As with some of the other guides, the application shall manage fruit entities.

```
package org.acme.neo4j;

public class Fruit {

    public Long id;

    public String name;

    public Fruit() {
        // This is needed for the REST-Easy JSON Binding
    }

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

    public Fruit(Long id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Prerequisites

To complete this guide, you need:

- JDK 1.8+ installed with JAVA_HOME configured appropriately

- an IDE

- Apache Maven 3.6.3

- Access to a Neo4j Database

- Optional Docker for your system

Setup Neo4j

The easiest way to start a Neo4j instance is a locally installed Docker environment.

```
docker run --publish=7474:7474 --publish=7687:7687 -e
'NEO4J_AUTH=neo4j/secret' neo4j:4.0.0
```

This starts a Neo4j instance, that publishes its Bolt port on 7687 and a web interface on http://localhost:7474.

Have a look at the download page for other options to get started with the product itself.

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.

Clone the Git repository: git clone https://github.com/quarkusio/quarkus-quickstarts.git, or download an archive.

The solution is located in the neo4j-quickstart directory. It contains a very simple UI to use the JAX-RS resources created here, too.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=neo4j-quickstart \
    -Dextensions="neo4j,resteasy-jsonb"
cd neo4j-quickstart
```

It generates:

- the Maven structure

- a landing page accessible on http://localhost:8080

- example Dockerfile files for both native and jvm modes

- the application configuration file

- an `org.acme.datasource.GreetingResource` resource

- an associated test

The Neo4j extension has been added already to your `pom.xml`. In addition, we added `resteasy-jsonb`, which allows us to expose `Fruit` instances over HTTP in the JSON format via JAX-RS resources. If you have an already created project, the `neo4j` extension can be added to an existing Quarkus project with the `add-extension` command:

```
./mvnw quarkus:add-extension -Dextensions="neo4j"
```

Otherwise, you can manually add this to the dependencies section of your `pom.xml` file:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-neo4j</artifactId>
</dependency>
```

Configuring

The Neo4j driver can be configured with standard Quarkus properties:

src/main/resources/application.properties

```
quarkus.neo4j.uri = bolt://localhost:7687
quarkus.neo4j.authentication.username = neo4j
quarkus.neo4j.authentication.password = secret
```

You'll recognize the authentication here that you passed on to the docker command above.

Having done that, the driver is ready to use, there are however other configuration options, detailed below.

Using the driver

General remarks

The result of a statement consists usually of one or more `org.neo4j.driver.Record`. Those records contain arbitrary values, supported by the driver. If you return a node of the graph, it will be a `org.neo4j.driver.types.Node`.

We add the following method to the `Fruit`, as a convenient way to create them:

```
public static Fruit from(Node node) {
    return new Fruit(node.id(), node.get("name").asString());
}
```

Add a FruitResource skeleton like this and @Inject a org.neo4j.driver.Driver instance:

src/main/java/org/acme/neo4j/FruitResource.java

```
package org.acme.neo4j;

import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.neo4j.driver.Driver;

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

    @Inject
    Driver driver;
}
```

Reading nodes

Add the following method to the fruit resource:

src/main/java/org/acme/neo4j/FruitResource.java

```java
@GET
public CompletionStage<Response> get() {
    AsyncSession session = driver.asyncSession(); ①
    return session
        .runAsync("MATCH (f:Fruit) RETURN f ORDER BY f.name")  ②
        .thenCompose(cursor ->  ③
            cursor.listAsync(record ->
Fruit.from(record.get("f").asNode()))
        )
        .thenCompose(fruits ->  ④
            session.closeAsync().thenApply(signal -> fruits)
        )
        .thenApply(Response::ok) ⑤
        .thenApply(ResponseBuilder::build);
}
```

① Open a new, asynchronous session with Neo4j

② Execute a query. This is a Cypher statement.

③ Retrieve a cursor, list the results and create `Fruit`s.

④ Close the session after processing

⑤ Create a JAX-RS response

Now start Quarkus in `dev` mode with:

```
./mvnw compile quarkus:dev
```

and retrieve the endpoint like this

```
curl localhost:8080/fruits
```

There are not any fruits, so let's create some.

Creating nodes

The `POST` method looks similar. It uses transaction functions of the driver:

src/main/java/org/acme/neo4j/FruitResource.java

```java
@POST
public CompletionStage<Response> create(Fruit fruit) {
    AsyncSession session = driver.asyncSession();
    return session
        .writeTransactionAsync(tx -> tx
            .runAsync("CREATE (f:Fruit {name: $name}) RETURN f",
Values.parameters("name", fruit.name))
            .thenCompose(fn -> fn.singleAsync())
        )
        .thenApply(record -> Fruit.from(record.get("f").asNode()))
        .thenCompose(persistedFruit ->
session.closeAsync().thenApply(signal -> persistedFruit))
        .thenApply(persistedFruit -> Response
            .created(URI.create("/fruits/" + persistedFruit.id))
            .build()
        );
}
```

As you can see, we are now using a Cypher statement with named parameters (The $name of the fruit). The node is returned, a Fruit entity created and then mapped to a 201 created response.

A curl request against this path may look like this:

```
curl -v -X "POST" "http://localhost:8080/fruits" \
     -H 'Content-Type: application/json; charset=utf-8' \
     -d $'{
  "name": "Banana"
}'
```

The response contains an URI that shall return single nodes.

Read single nodes

This time, we ask for a read-only transaction. We also add some exception handling, in case the resource is called with an invalid id:

```java
@GET
@Path("{id}")
public CompletionStage<Response> getSingle(@PathParam("id") Long
id) {
    AsyncSession session = driver.asyncSession();
    return session
        .readTransactionAsync(tx -> tx
            .runAsync("MATCH (f:Fruit) WHERE id(f) = $id RETURN f",
Values.parameters("id", id))
            .thenCompose(fn -> fn.singleAsync())
    )
    .handle((record, exception) -> {
        if(exception != null) {
            Throwable source = exception;
            if(exception instanceof CompletionException) {
                source =
((CompletionException)exception).getCause();
            }
            Status status = Status.INTERNAL_SERVER_ERROR;
            if(source instanceof NoSuchRecordException) {
                status = Status.NOT_FOUND;
            }
            return Response.status(status).build();
        } else  {
            return
Response.ok(Fruit.from(record.get("f").asNode())).build();
        }
    })
    .thenCompose(response -> session.closeAsync().thenApply(signal
-> response));
}
```

A request may look like this:

```
curl localhost:8080/fruits/42
```

> 🛈 In case Neo4j has been setup as a cluster, the transaction mode is used to decide whether a request is routed to a leader or a follower instance. Write transactions must be handled by a leader, whereas read-only transactions can be handled by followers.

Deleting nodes

Finally, we want to get rid of fruits again and we add the DELETE method:

src/main/java/org/acme/neo4j/FruitResource.java

```
@DELETE
@Path("{id}")
public CompletionStage<Response> delete(@PathParam("id") Long id) {

    AsyncSession session = driver.asyncSession();
    return session
        .writeTransactionAsync(tx -> tx
            .runAsync("MATCH (f:Fruit) WHERE id(f) = $id DELETE f",
Values.parameters("id", id))
            .thenCompose(fn -> fn.consumeAsync()) ①
        )
        .thenCompose(response -> session.closeAsync())
        .thenApply(signal -> Response.noContent().build());
}
```

① There is no result for us, only a summary of the query executed.

A request may look like this

```
curl -X DELETE localhost:8080/fruits/42
```

And that's already the most simple CRUD application with one type of nodes. Feel free to add relationships to the model. One idea would be to model recipes that contain fruits. The Cypher manual linked in the introduction will help you with modelling your queries.

Next steps

Packaging

Packaging your application is as simple as `./mvnw clean package`. It can be run with `java -jar target/neo4j-quickstart-1.0-SNAPSHOT-runner.jar`.

With GraalVM installed, you can also create a native executable binary: `./mvnw clean package -Dnative`. Depending on your system, that will take some time.

Connection Health Check

If you are using the `quarkus-smallrye-health` extension, `quarkus-neo4j` will automatically add a readiness health check to validate the connection to Neo4j.

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.neo4j.health.enabled` property to `false` in your `application.properties`.

Driver metrics

If you are using a metrics extension and specify the config property `quarkus.neo4j.pool.metrics-enabled=true`, the Neo4j extension will expose several metrics related to the Neo4j connection pool.

Explore Cypher and the Graph

There are tons of options to model your domain within a Graph. The Neo4j docs, the sandboxes and more are a good starting point.

Going reactive

If you have access to Neo4j 4.0, you can go fully reactive.

To make life a bit easier, we will use Mutiny for this.

> *Mutiny*
>
> The following example uses Mutiny reactive types, if you're not familiar with them, read the Getting Started with Reactive guide first.

Add the following dependency to your `pom.xml`:

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

The reactive fruit resources streams the name of all fruits:

src/main/java/org/acme/neo4j/ReactiveFruitResource.java

```java
package org.acme.neo4j;

import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.Uni;
import org.neo4j.driver.Driver;
import org.neo4j.driver.reactive.RxResult;
import org.reactivestreams.Publisher;


@Path("reactivefruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ReactiveFruitResource {

    @Inject
    Driver driver;

    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS)
    public Publisher<String> get() {
        // Create a stream from a resource we can close in a
finalizer:
        return Multi.createFrom().resource(driver::rxSession,
                session -> session.readTransaction(tx -> {
                    RxResult result = tx.run("MATCH (f:Fruit)
RETURN f.name as name ORDER BY f.name");
                    return
Multi.createFrom().publisher(result.records())
                            .map(record ->
record.get("name").asString());
                })
        ).withFinalizer(session -> {
            return Uni.createFrom().publisher(session.close());
        });
    }
}
```

`driver.rxSession()` returns a reactive session. It exposes its API based on Reactive Streams, most prominently, as `org.reactivestreams.Publisher`. Those can be used directly, but we found it easier and more expressive to wrap them in reactive types such as the one provided by

Mutiny. Typically, in the previous code, the session is closed when the stream completes, fails or the subscriber cancels.

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.neo4j.health.enabled` Whether or not an health check is published in case the smallrye-health extension is present.	boolean	`true`
`quarkus.neo4j.uri` The uri this driver should connect to. The driver supports bolt, bolt+routing or neo4j as schemes.	string	`bolt:/ /local host:7 687`
`quarkus.neo4j.encrypted` If the driver should use encrypted traffic.	boolean	`false`
Authentication	**Type**	**Default**
`quarkus.neo4j.authentication.username` The login of the user connecting to the database.	string	`neo4j`
`quarkus.neo4j.authentication.password` The password of the user connecting to the database.	string	`neo4j`
`quarkus.neo4j.authentication.disabled` Set this to true to disable authentication.	boolean	`false`
Configure trust settings for encrypted traffic	**Type**	**Default**

	Type	Default
`quarkus.neo4j.trust-settings.strategy` Configures which trust strategy to apply when using encrypted traffic.	`trust-all-certificates`, `trust-custom-ca-signed-certificates`, `trust-system-ca-signed-certificates`	`trust-system-ca-signed-certificates`
`quarkus.neo4j.trust-settings.cert-file` The file of the certificate to use.	path	
`quarkus.neo4j.trust-settings.hostname-verification-enabled` If hostname verification is used.	boolean	`false`
Connection pool	**Type**	**Default**
`quarkus.neo4j.pool.metrics-enabled` Flag, if metrics are enabled.	boolean	`false`
`quarkus.neo4j.pool.log-leaked-sessions` Flag, if leaked sessions logging is enabled.	boolean	`false`
`quarkus.neo4j.pool.max-connection-pool-size` The maximum amount of connections in the connection pool towards a single database.	int	`100`
`quarkus.neo4j.pool.idle-time-before-connection-test` Pooled connections that have been idle in the pool for longer than this timeout will be tested before they are used again. The value `0` means connections will always be tested for validity and negative values mean connections will never be tested.	Duration ❓	`-0.001S`

`quarkus.neo4j.pool.max-connection-lifetime` Pooled connections older than this threshold will be closed and removed from the pool.	Duration ❓	1H
`quarkus.neo4j.pool.connection-acquisition-timeout` Acquisition of new connections will be attempted for at most configured timeout.	Duration ❓	1M

ℹ️ *About the Duration format*

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.