neo4j

The Neo4j Java Driver Manual v5.0

Table of Contents

Quickstart

The Neo4j Java driver is the official library to interact with a Neo4j instance through a Java application.

At the hearth of Neo4j lies Cypher, the query language to interact with a Neo4j database. While this guide does not *require* you to be a seasoned Cypher querier, it is going to be easier to focus on the Java-specific bits if you already know some Cypher. For this reason, although this guide does *also* provide a gentle introduction to Cypher along the way, consider checking out Getting started → Cypher for a more detailed walkthrough of graph databases modelling and querying if this is your first approach. You may then apply that knowledge while following this guide to develop your Java application.

Installation

Add the Neo4j Java driver to the list of dependencies in the `pom.xml` of your Maven project:

```
<dependency>
    <groupId>org.neo4j.driver</groupId>
    <artifactId>neo4j-java-driver</artifactId>
    <version>5.0</version>
</dependency>
```

More info on installing the driver →

Connect to the database

Connect to a database by creating a Driver object and providing a URL and an authentication token. Once you have a `Driver` instance, use the `.verifyConnectivity()` method to ensure that a working connection can be established.

```
package demo;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;

public class App {

    public static void main(String... args) {

        // URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();
            System.out.println("Connection established.");
        }
    }
}
```

More info on connecting to a database →

Query the database

Execute a Cypher statement with the method `Driver.executableQuery()`. Do not hardcode or concatenate parameters: use placeholders and specify the parameters as a map through the `.withParameters()` method.

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

// Get all 42-year-olds
var result = driver.executableQuery("MATCH (p:Person {age: $age}) RETURN p.name AS name")
    .withParameters(Map.of("age", 42))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();

// Loop through results and do something with them
var records = result.records();
records.forEach(r -> {
    System.out.println(r);  // or r.get("name").asString()
});

// Summary information
var summary = result.summary();
System.out.printf("The query %s returned %d records in %d ms.%n",
    summary.query(), records.size(),
    summary.resultAvailableAfter(TimeUnit.MILLISECONDS));
```

More info on querying the database →

Run your own transactions

For more advanced use-cases, you can run transactions. Use the methods `Session.executeRead()` and `Session.executeWrite()` to run managed transactions.

A transaction with multiple queries, client logic, and potential roll backs

```
package demo;

import java.util.Map;
import java.util.List;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;
import org.neo4j.driver.Record;
import org.neo4j.driver.RoutingControl;
import org.neo4j.driver.SessionConfig;
import org.neo4j.driver.TransactionContext;
import org.neo4j.driver.exceptions.NoSuchRecordException;

public class App {

    // Create & employ 100 people to 10 different organizations
    public static void main(String... args) {

        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
                for (int i=0; i<100; i++) {
                    String name = String.format("Thor%d", i);
                    try {
                        String orgId = session.executeWrite(tx -> employPersonTx(tx, name));
                        System.out.printf("User %s added to organization %s.%n", name, orgId);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                }
            }
        }
    }

    static String employPersonTx(TransactionContext tx, String name) {
        final int employeeThreshold = 10;

        // Create new Person node with given name, if not exists already
        tx.run("MERGE (p:Person {name: $name})", Map.of("name", name));

        // Obtain most recent organization ID and the number of people linked to it
        var result = tx.run("""
            MATCH (o:Organization)
            RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employeesN
            ORDER BY o.createdDate DESC
            LIMIT 1
            """);

        Record org = null;
        String orgId = null;
        int employeesN = 0;
        try {
            org = result.single();
            orgId = org.get("id").asString();
            employeesN = org.get("employeesN").asInt();
        } catch (NoSuchRecordException e) {
            // The query is guaranteed to return <= 1 results, so if.single() throws, it means there's
none.
            // If no organization exists, create one and add Person to it
            orgId = createOrganization(tx);
            System.out.printf("No orgs available, created %s.%n", orgId);
        }

        // If org does not have too many employees, add this Person to it
        if (employeesN < employeeThreshold) {
            addPersonToOrganization(tx, name, orgId);
            // If the above throws, the transaction will roll back
            // -> not even Person is created!

        // Otherwise, create a new Organization and link Person to it
        } else {
            orgId = createOrganization(tx);
            System.out.printf("Latest org is full, created %s.%n", orgId);
            addPersonToOrganization(tx, name, orgId);
            // If any of the above throws, the transaction will roll back
            // -> not even Person is created!
        }

        return orgId;  // Organization ID to which the new Person ends up in
    }

    static String createOrganization(TransactionContext tx) {
        var result = tx.run("""
            CREATE (o:Organization {id: randomuuid(), createdDate: datetime()})
            RETURN o.id AS id
        """);
        var org = result.single();
        var orgId = org.get("id").asString();
        return orgId;
    }

    static void addPersonToOrganization(TransactionContext tx, String personName, String orgId) {
        tx.run("""
            MATCH (o:Organization {id: $orgId})
            MATCH (p:Person {name: $name})
            MERGE (p)-[:WORKS_FOR]->(o)
            """, Map.of("orgId", orgId, "name", personName)
        );
    }
}
```

More info on running transactions →

Close connections and sessions

Unless you created them with `try-with-resources` statements, call the `.close()` method on all `Driver` and `Session` instances to release any resources still held by them.

```
session.close();
driver.close();
```

API documentation

For in-depth information about driver features, check out the API documentation.

Glossary

LTS

A *Long Term Support* release is one guaranteed to be supported for a number of years. Neo4j 4.4 is LTS, and Neo4j 5 will also have an LTS version.

Aura

Aura is Neo4j's fully managed cloud service. It comes with both free and paid plans.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the database. It is like SQL, but for graphs.

APOC

Awesome Procedures On Cypher (APOC) is a library of (many) functions that can not be easily expressed in Cypher itself.

Bolt

Bolt is the protocol used for interaction between Neo4j instances and drivers. It listens on port 7687 by default.

ACID

Atomicity, Consistency, Isolation, Durability (ACID) are properties guaranteeing that database transactions are processed reliably. An ACID-compliant DBMS ensures that the data in the database remains accurate and consistent despite failures.

eventual consistency

A database is eventually consistent if it provides the guarantee that all cluster members will, *at some point in time*, store the latest version of the data.

causal consistency

A database is causally consistent if read and write queries are seen by every member of the cluster in the same order. This is stronger than *eventual consistency*.

NULL

The null marker is not a type but a placeholder for absence of value. For more information, see Cypher → Working with `null`.

transaction

A transaction is a unit of work that is either *committed* in its entirety or *rolled back* on failure. An example is a bank transfer: it involves multiple steps, but they must *all* succeed or be reverted, to avoid money being subtracted from one account but not added to the other.

backpressure

Backpressure is a force opposing the flow of data. It ensures that the client is not being overwhelmed by data faster than it can handle.

transaction function

A transaction function is a callback executed by an `executeRead` or `executeWrite` call. The driver automatically re-executes the callback in case of server failure.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database.

=Regular workflow=

Installation

To start creating a Neo4j Java application, you first need to install the Java driver and get a Neo4j database instance to connect to.

> The driver requires Java 17 or higher.

Install the driver

If you already have a Maven project, you may add the driver as a dependency in your `pom.xml` file:

```
<dependency>
    <groupId>org.neo4j.driver</groupId>
    <artifactId>neo4j-java-driver</artifactId>
    <version>5.0</version>
</dependency>
```

How to create a Maven project for a Neo4j Java application?

If you are new to Maven, you may download the Neo4j demo app example project and use it as a base to experiment with the driver and build your application.

The file `App.java` provides a skeleton application. You may compile and run the project with the commands:

```
mvn install
java -jar target/neo4j-demo-app-1.0-SNAPSHOT-jar-with-dependencies.jar
```

For other dependency management systems, refer to the driver's package page.

Always use the latest version of the driver, as it will always work both with the previous Neo4j LTS release and with the current and next major releases. The latest `5.x` driver supports connection to any Neo4j 5 and 4.4 instance, and will also be compatible with Neo4j 6. For a detailed list of changes across versions, see the driver's changelog.

Get a Neo4j instance

You need a running Neo4j database in order to use the driver with it. The easiest way to spin up a **local instance** is through a Docker container (requires `docker.io`). The command below runs the latest Neo4j version in Docker, setting the admin username to `neo4j` and password to `secretgraph`:

```
docker run \
   -p7474:7474 \                        # forward port 7474 (HTTP)
   -p7687:7687 \                        # forward port 7687 (Bolt)
   -d \                                 # run in background
   -e NEO4J_AUTH=neo4j/secretgraph \    # set login credentials
   neo4j:latest
```

Alternatively, you can obtain a free **cloud instance** through Aura.

You can also install Neo4j on your system, or use Neo4j Desktop to create a local development environment (not for production).

Connection

Once you have installed the driver and have a running Neo4j instance, you are ready to connect your application to the database.

Connect to the database

You connect to a database by creating a Driver object and providing a URL and an authentication token.

```
package demo;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;

public class App {

    public static void main(String... args) {

        // URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {  ①
            driver.verifyConnectivity();  ②
            System.out.println("Connection established.");
        }
    }
}
```

① Creating a `Driver` instance only provides information on *how* to access the database, but does not actually *establish* a connection. Connection is instead deferred to when the first query is executed.

② To verify immediately that the driver can connect to the database (valid credentials, compatible versions, etc), use the `.verifyConnectivity()` method after initializing the driver.

Both the creation of a `Driver` object and the connection verification can raise a number of different exceptions. Since a connection error is a blocker for any subsequent task, the most common choice is to let the program crash should an exception occur while estabilishing a connection.

`Driver` objects are immutable, thread-safe, and expensive to create, so your application should create only one instance and pass it around (you may share `Driver` instances across threads). If you need to query the database through several different users, use impersonation without creating a new `Driver` instance. If you want to alter a `Driver` configuration, you need to create a new object.

Connect to an Aura instance

When you create an Aura instance, you may download a text file (a so-called *Dotenv file*) containing the connection information to the database in the form of environment variables. The file has a name of the form `Neo4j-a0a2fa1d-Created-2023-11-06.txt`.

You can either manually extract the URI and the credentials from that file, or use a third party-module to load them. We recommend the module `dotenv-java` for that purpose.

Using `dotenv-java` to extract credentials from a Dotenv file

```java
package demo;

import io.github.cdimascio.dotenv.Dotenv;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;

public class App {

    public static void main(String... args) {

        var dotenv = Dotenv.configure()
            //.directory("/path/to/env/file")
            .filename("Neo4j-a0a2fa1d-Created-2023-11-06.txt")
            .load();

        final String dbUri = dotenv.get("NEO4J_URI");
        final String dbUser = dotenv.get("NEO4J_USERNAME");
        final String dbPassword = dotenv.get("NEO4J_PASSWORD");

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();
             System.out.println("Connection established.");
        }
    }
}
```

An Aura instance is not conceptually different from any other Neo4j instance, as Aura is simply a *deployment mode* for Neo4j. When interacting with a Neo4j database through the driver, it doesn't make a difference whether it is an Aura instance it is working with or a different deployment.

Close connections

Always close `Driver` objects to free up all allocated resources, even upon unsuccessful connection or runtime errors. Either create the `Driver` object using the `try-with-resources` statement, or call the `Driver.close()` method explicitly.

Further connection parameters

For more `Driver` configuration parameters and further connection settings, see Advanced connection information.

Query the database

Once you have connected to the database, you can execute Cypher queries through the method `Driver.executableQuery()`.

> `Driver.executableQuery()` was introduced with the version 5.8 of the driver. For queries with earlier versions, use sessions and transactions.

Write to the database

To create a node representing a person named `Alice`, use the Cypher clause `CREATE`:

Create a node representing a person named `Alice`

```
// import java.util.Map;
// import java.util.concurrent.TimeUnit;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("CREATE (:Person {name: $name})")  ①
    .withParameters(Map.of("name", "Alice"))  ②
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())  ③
    .execute();
var summary = result.summary();  ④
System.out.printf("Created %d records in %d ms.%n",
    summary.counters().nodesCreated(),
    summary.resultAvailableAfter(TimeUnit.MILLISECONDS));
```

① The Cypher query

② A map of query parameters

③ Which database the query should be run against

④ The summary of execution returned by the server

Read from the database

To retrieve information from the database, use the Cypher clause `MATCH`:

Retrieve all `Person` *nodes*

```
// import java.util.concurrent.TimeUnit;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name AS name")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();

// Loop through results and do something with them
var records = result.records();  ①
records.forEach(r -> {
    System.out.println(r);  // or r.get("name").asString()
});

// Summary information
var summary = result.summary();  ②
System.out.printf("The query %s returned %d records in %d ms.%n",
    summary.query(), records.size(),
    summary.resultAvailableAfter(TimeUnit.MILLISECONDS));
```

① `records` contains the result as a list of `Record` objects

② `summary` contains the summary of execution returned by the server

Properties inside a `Record` object are embedded within `Value` objects. To extract and cast them to the corresponding Java types, use `.as<type>()` (eg. `.asString()`, `asInt()`, etc). For example, if the `name` property coming from the database is a string, `record.get("name").asString()` will yield the property value as a `String` object.

For more information, see Data types and mapping to Cypher types.

Update the database

To update a node's information in the database, use the Cypher clauses `SET`:

Update node `Alice` to add an `age` property

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("""
    MATCH (p:Person {name: $name})
    SET p.age = $age
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .withParameters(Map.of("name", "Alice", "age", 42))
    .execute();
var summary = result.summary();
System.out.println("Query updated the database?");
System.out.println(summary.counters().containsUpdates());
```

To create a new relationship, linking it to two already existing node, use a combination of the Cypher clauses `MATCH` and `CREATE`:

Create a relationship `:KNOWS` between `Alice` and `Bob`

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("""
    MATCH (alice:Person {name: $name})  ①
    MATCH (bob:Person {name: $friend})  ②
    CREATE (alice)-[:KNOWS]->(bob)  ③
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .withParameters(Map.of("name", "Alice", "friend", "Bob"))
    .execute();
var summary = result.summary();
System.out.println("Query updated the database?");
System.out.println(summary.counters().containsUpdates());
```

① Retrieve the person node named `Alice` and bind it to a variable `alice`

② Retrieve the person node named `Bob` and bind it to a variable `bob`

③ Create a new `:KNOWS` relationship outgoing from the node bound to `alice` and attach to it the `Person` node named `Bob`

Delete from the database

To remove a node and any relationship attached to it, use the Cypher clause `DETACH DELETE`:

Remove the `Alice` node and all its relationships

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

// This does not delete _only_ p, but also all its relationships!
var result = driver.executableQuery("""
    MATCH (p:Person {name: $name})
    DETACH DELETE p
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .withParameters(Map.of("name", "Alice"))
    .execute();
var summary = result.summary();
System.out.println("Query updated the database?");
System.out.println(summary.counters().containsUpdates());
```

Query parameters

Do not hardcode or concatenate parameters directly into queries. Instead, always use placeholders and specify the Cypher parameters, as shown in the previous examples. This is for:

1. **performance benefits**: Neo4j compiles and caches queries, but can only do so if the query structure is unchanged;
2. **security reasons**: see protecting against Cypher injection.

You may provide query parameters as a map through the `.withParameters()` method.

```
var result = driver.executableQuery("MATCH (p:Person {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

There can be circumstances where your query structure prevents the usage of parameters in all its parts. For those rare use cases, see Dynamic values in property keys, relationship types, and labels.

Error handling

A query run may fail for a number of reasons, with different exceptions being raised. Some of them are due to Cypher syntax errors, permission issues, or other forms of misconfiguration/misusage. The choice of how to handle those exceptions is up to your application: whether you want to be defensive (for example check if there are records to process to avoid NoSuchRecordException), or whether you want to catch and handle the exceptions as they arise.

The driver automatically retries to run a failed query, if the failure is deemed to be transient (for example due to temporary server unavailability). An exception will be raised if the operation keeps failing after a number of attempts.

Query configuration

You can supply further configuration parameters to alter the default behavior of `.executableQuery()`. You do so through the method `.withConfig()`, which takes a `QueryConfig` object.

Database selection

It is recommended to **always specify the database explicitly** with the `.withDatabase("<dbName>")` method, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database. If no database is given, the user's home database is used.

```
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

Specifying the database through the configuration method is preferred over the `USE` Cypher clause. If the server runs on a cluster, queries with `USE` require server-side routing to be enabled. Queries may also take longer to execute as they may not reach the right cluster member at the first attempt, and need to be routed to one containing the requested database.

Request routing

In a cluster environment, all queries are directed to the leader node by default. To improve performance on read queries, you can use the method `.withRouting(RoutingControl.READ)` to route a query to the read nodes.

```
// import org.neo4j.driver.QueryConfig;
// import org.neo4j.driver.RoutingControl;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name")
    .withConfig(QueryConfig.builder()
        .withDatabase("neo4j")
        .withRouting(RoutingControl.READ)
        .build())
    .execute();
```

Although executing a *write* query in read mode likely results in a runtime error, **you should not rely on this for access control.** The difference between the two modes is that *read* transactions will be routed to any node of a cluster, whereas *write* ones will be directed to the leader. In other words, there is no guarantee that a write query submitted in read mode will be rejected.

Run queries as a different user

You can execute a query under the security context of a different user with the method `.withImpersonatedUser("<username>")`, specifying the name of the user to impersonate. For this to work, the user under which the `Driver` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `Driver` object.

```java
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name")
    .withConfig(QueryConfig.builder()
        .withDatabase("neo4j")
        .withImpersonatedUser("somebodyElse")
        .build())
    .execute();
```

When impersonating a user, the query is run within the complete security context of the impersonated user and not the authenticated user (i.e. home database, permissions, etc.).

A full example

```java
package demo;

import java.util.Map;
import java.util.List;
import java.util.concurrent.TimeUnit;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Record;
import org.neo4j.driver.QueryConfig;
import org.neo4j.driver.RoutingControl;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {

            List<Map> people = List.of(
                Map.of("name", "Alice", "age", 42, "friends", List.of("Bob", "Peter", "Anna")),
                Map.of("name", "Bob", "age", 19),
                Map.of("name", "Peter", "age", 50),
                Map.of("name", "Anna", "age", 30)
            );

            try {

                //Create some nodes
                people.forEach(person -> {
                    var result = driver.executableQuery("CREATE (p:Person {name: $person.name, age:
$person.age})")
                        .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                        .withParameters(Map.of("person", person))
                        .execute();
                });

                // Create some relationships
                people.forEach(person -> {
                    if(person.containsKey("friends")) {
                        var result = driver.executableQuery("""
                            MATCH (p:Person {name: $person.name})
                            UNWIND $person.friends AS friend_name
                            MATCH (friend:Person {name: friend_name})
                            CREATE (p)-[:KNOWS]->(friend)
                             """)
                            .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                            .withParameters(Map.of("person", person))
                            .execute();
                    }
                });

                // Retrieve Alice's friends who are under 40
                var result = driver.executableQuery("""
                    MATCH (p:Person {name: $name})-[:KNOWS]-(friend:Person)
                    WHERE friend.age < $age
                    RETURN friend
                     """)
                    .withConfig(QueryConfig.builder()
                        .withDatabase("neo4j")
                        .withRouting(RoutingControl.READ)
                        .build())
                    .withParameters(Map.of("name", "Alice", "age", 40))
                    .execute();

                // Loop through results and do something with them
                result.records().forEach(r -> {
                    System.out.println(r);
                });

                // Summary information
                System.out.printf("The query %s returned %d records in %d ms.%n",
                    result.summary().query(), result.records().size(),
                    result.summary().resultAvailableAfter(TimeUnit.MILLISECONDS));

            } catch (Exception e) {
                System.out.println(e.getMessage());
                System.exit(1);
            }
        }
    }
}
```

For more information see API documentation → Driver.executableQuery().

=Advanced usage=

Run your own transactions

When querying the database with `executableQuery()`, the driver automatically creates a *transaction*. A transaction is a unit of work that is either *committed* in its entirety or *rolled back* on failure. You can include multiple Cypher statements in a single query, as for example when using `MATCH` and `CREATE` in sequence to update the database, but you cannot have multiple queries and interleave some client-logic in between them.

For these more advanced use-cases, the driver provides functions to take full control over the transaction lifecycle. These are called *managed transactions*, and you can think of them as a way of unwrapping the flow of `executableQuery()` and being able to specify its desired behavior in more places.

Create a session

Before running a transaction, you need to obtain a *session*. Sessions act as concrete query channels between the driver and the server, and ensure causal consistency is enforced.

Sessions are created with the method `Driver.session()`. Use the optional argument to alter the session's configuration, among which for example the target database. For further configuration parameters, see Session configuration.

```
// import org.neo4j.driver.SessionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    // session usage
}
```

Session creation is a lightweight operation, so sessions can be created and destroyed without significant cost. Always close sessions when you are done with them.

Sessions are not thread safe: you can share the main `Driver` object across threads, but make sure each thread creates its own sessions.

Run a managed transaction

A transaction can contain any number of queries. As Neo4j is ACID compliant, **queries within a transaction will either be executed as a whole or not at all**: you cannot get a part of the transaction succeeding and another failing. Use transactions to group together related queries which work together to achieve a single *logical* database operation.

A managed transaction is created with the methods `Session.executeRead()` and `Session.executeWrite()`, depending on whether you want to retrieve data from the database or alter it. Both methods take a transaction function callback, which is responsible for actually carrying out the queries and processing the result.

Retrieve people whose name starts with `Al`.

```
// import java.util.Map
// import org.neo4j.driver.SessionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {  ①
    var people = session.executeRead(tx -> {  ②
        var result = tx.run("""
        MATCH (p:Person) WHERE p.name STARTS WITH $filter  ③
        RETURN p.name AS name ORDER BY name
        """, Map.of("filter", "Al"));
        return result.list();  // return a list of Record objects ④
    });
    people.forEach(person -> {
        System.out.println(person);
    });
    // further tx.run() calls will execute within the same transaction
}
```

① Create a session. A single session can be the container for multiple queries. Unless created as a resource using the `try` construct, remember to close it when done.

② The `.executeRead()` (or `.executeWrite()`) method is the entry point into a transaction. It takes a callback to a *transaction function*, which is responsible of running queries.

③ Use the method `tx.run()` to execute queries. You can provide a map of query parameters as second argument. Each query run returns a `Result` object.

④ Process the result using any of the methods on `Result`. The method `.list()` retrieves all records into a list.

Do not hardcode or concatenate parameters directly into the query. Use query parameters instead, both for performance and security reasons.

Transaction functions should never return the `Result` object directly. Instead, always process the result in some way. Within a transaction function, a `return` statement results in the transaction being committed, while the transaction is automatically rolled back if an exception is raised.

The methods `.executeRead()` and `.executeWrite()` have replaced `.readTransaction()` and `.writeTransaction()`, which are deprecated in version 5.x and will be removed in version 6.0.

A transaction with multiple queries, client logic, and potential roll backs

```
package demo;

import java.util.Map;
import java.util.List;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;
import org.neo4j.driver.Record;
import org.neo4j.driver.RoutingControl;
import org.neo4j.driver.SessionConfig;
import org.neo4j.driver.TransactionContext;
import org.neo4j.driver.exceptions.NoSuchRecordException;

public class App {

    // Create & employ 100 people to 10 different organizations
    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
                for (int i=0; i<100; i++) {
                    String name = String.format("Thor%d", i);

                    try {
                        String orgId = session.executeWrite(tx -> employPersonTx(tx, name));
                        System.out.printf("User %s added to organization %s.%n", name, orgId);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                }
            }
        }
    }

    static String employPersonTx(TransactionContext tx, String name) {
        final int employeeThreshold = 10;

        // Create new Person node with given name, if not exists already
        tx.run("MERGE (p:Person {name: $name})", Map.of("name", name));

        // Obtain most recent organization ID and the number of people linked to it
        var result = tx.run("""
            MATCH (o:Organization)
            RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employeesN
            ORDER BY o.createdDate DESC
            LIMIT 1
            """);

        Record org = null;
        String orgId = null;
        int employeesN = 0;
        try {
            org = result.single();
            orgId = org.get("id").asString();
            employeesN = org.get("employeesN").asInt();
        } catch (NoSuchRecordException e) {
            // The query is guaranteed to return <= 1 results, so if.single() throws, it means there's
none.
            // If no organization exists, create one and add Person to it
            orgId = createOrganization(tx);
            System.out.printf("No orgs available, created %s.%n", orgId);
        }

        // If org does not have too many employees, add this Person to it
        if (employeesN < employeeThreshold) {
            addPersonToOrganization(tx, name, orgId);
            // If the above throws, the transaction will roll back
            // -> not even Person is created!

        // Otherwise, create a new Organization and link Person to it
        } else {
            orgId = createOrganization(tx);
            System.out.printf("Latest org is full, created %s.%n", orgId);
            addPersonToOrganization(tx, name, orgId);
            // If any of the above throws, the transaction will roll back
            // -> not even Person is created!
        }

        return orgId;  // Organization ID to which the new Person ends up in
    }

    static String createOrganization(TransactionContext tx) {
        var result = tx.run("""
            CREATE (o:Organization {id: randomuuid(), createdDate: datetime()})
            RETURN o.id AS id
        """);
        var org = result.single();
        var orgId = org.get("id").asString();
        return orgId;
    }
    static void addPersonToOrganization(TransactionContext tx, String personName, String orgId) {
        tx.run("""
            MATCH (o:Organization {id: $orgId})
            MATCH (p:Person {name: $name})
            MERGE (p)-[:WORKS_FOR]->(o)
            """, Map.of("orgId", orgId, "name", personName)
        );
    }
}
```

Should a transaction fail for a reason that the driver deems transient, it automatically retries to run the transaction function (with an exponentially increasing delay). For this reason, **transaction functions must be *idempotent*** (i.e., they should produce the same effect when run several times), because you do not know upfront how many times they are going to be executed. In practice, this means that you should not edit nor rely on globals, for example. Note that although transactions functions might be executed multiple times, the queries inside it will always run only once.

A session can chain multiple transactions, but **only one single transaction can be active within a session at any given time.** To maintain multiple concurrent transactions, use multiple concurrent sessions.

The transaction functions callback passed to `.executeRead()` and `.executeWrite()` may return anything, as the return type is a Java Generic. **That also means they may not return `void`**, as that is not an instance of `Object`.
If you are not interested in returning anything out of a transaction function, you may either:

1. use `.executeWriteWithoutResult()`, which supports returning `void` (and `void` only);
2. use `.executeRead()`/`.executeWrite()` and return `null` in the transaction function.

Run an explicit transaction

You can achieve full control over transactions by manually beginning one with the method `Session.beginTransaction()`, which returns a `Transaction` object. You may then run queries inside an explicit transaction with the method `Transaction.run()`.

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    try (Transaction tx = session.beginTransaction()) {
        // use tx.run() to run queries
        //     tx.commit() to commit the transaction
        //     tx.rollback() to rollback the transaction
    }
}
```

An explicit transaction can be committed with `Transaction.commit()` or rolled back with `Transaction.rollback()`. If no explicit action is taken, the driver will automatically roll back the transaction at the end of its lifetime.

Explicit transactions are most useful for applications that need to distribute Cypher execution across multiple functions for the same transaction, or for applications that need to run multiple queries within a single transaction but without the automatic retries provided by managed transactions.

An explicit transaction example involving an external API

```java
package demo;

import java.util.Map;
import java.util.List;
import java.util.Arrays;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;
import org.neo4j.driver.Record;
import org.neo4j.driver.SessionConfig;
import org.neo4j.driver.Transaction;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            String customerId = createCustomer(driver);
            int otherBankId = 42;
            transferToOtherBank(driver, customerId, otherBankId, 999);
        }
    }

    static String createCustomer(Driver driver) {
        var result = driver.executableQuery("""
            MERGE (c:Customer {id: randomUUID(), balance: 1000})
            RETURN c.id AS id
            """)
            .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
            .execute();
        return result.records().get(0).get("id").asString();
    }

    static void transferToOtherBank(Driver driver, String customerId, int otherBankId, float amount) {
        try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
            try (var tx = session.beginTransaction()) {
                if (! customerBalanceCheck(tx, customerId, amount)) {
                    System.out.printf("Customer %s doesn't have enough funds.%n", customerId);
                    return;  // give up
                }

                otherBankTransferApi(customerId, otherBankId, amount);
                // Now the money has been transferred => can't rollback anymore
                // (cannot rollback external services interactions)

                try {
                    decreaseCustomerBalance(tx, customerId, amount);
                    tx.commit();
                    System.out.printf("Transferred %f to %s.%n", amount, customerId);
                } catch (Exception e) {
                    requestInspection(customerId, otherBankId, amount, e);
                    throw new RuntimeException(e.getMessage());
                }
            }
        }
    }

    static boolean customerBalanceCheck(Transaction tx, String customerId, float amount) {
        var result = tx.run("""
            MATCH (c:Customer {id: $id})
            RETURN c.balance >= $amount AS sufficient
            """, Map.of("id", customerId, "amount", amount));
        var record = result.single();
        return record.get("sufficient").asBoolean();
    }

    static void otherBankTransferApi(String customerId, int otherBankId, float amount) {
        // make some API call to other bank
    }
    static void decreaseCustomerBalance(Transaction tx, String customerId, float amount) {
        tx.run("""
            MATCH (c:Customer {id: $id})
            SET c.balance = c.balance - $amount
            """, Map.of("id", customerId, "amount", amount));
    }

    static void requestInspection(String customerId, int otherBankId, float amount, Exception e) {
        // manual cleanup required; log this or similar
        System.out.printf("WARNING: transaction rolled back due to exception: %s.%n", e.getMessage());
        System.out.printf("customerId: %s, otherBankId: %d, amount: %f.%n", customerId, otherBankId,
amount);
    }
}
```

Process query results

The driver's output of a query is a `Result` object, which encapsulates the Cypher result in a rich data structure that requires some parsing on the client side. There are two main points to be aware of:

- **The result records are not immediately and entirely fetched and returned by the server**. Instead, results come as a *lazy stream*. In particular, when the driver receives some records from the server, they are initially *buffered* in a background queue. Records stay in the buffer until they are *consumed* by the application, at which point they are *removed from the buffer*. When no more records are available, the result is *exhausted*.
- **The result acts as a cursor**. This means that there is no way to retrieve a previous record from the stream, unless you saved it in an auxiliary data structure.

The animation below follows the path of a single query: it shows how the driver works with result records and how the application should handle results.

```
<video
  class="rounded-corners"
  controls
  width="100%"
  src="../../../common-content/5/_images/result.mp4"
  poster="../../../common-content/5/_images/result-poster.jpg"
  type="video/mp4"></video>
```

The easiest way of processing a result is by calling `.list()` on it, which yields a list of `Record` objects. Otherwise, a `Result` object implements a number of methods for processing records. The most commonly needed ones are listed below.

Method	Description
`list() List<Record>`	Return the remainder of the result as a list.
`single() Record`	Return the next and only remaining record. Calling this method always exhausts the result. If more (or less) than one record is available, a `NoSuchRecordException` is raised.
`next() Record`	Return the next record in the result. Throws `NoSuchRecordException` if no further records are available.

Method	Description
`hasNext() boolean`	Whether the result iterator has a next record to move to.
`peek() Record`	Return the next record from the result *without consuming it*. This leaves the record in the buffer for further processing.
`consume() ResultSummary`	Return the query result summary. It exhausts the result, so should only be called when data processing is over.

For a complete list of `Result` methods, see API documentation → Result.

Properties inside a `Record` object are embedded within `Value` objects. To extract and cast them to the corresponding Java types, use `.as<type>()` (eg. `.asString()`, `asInt()`, etc). For example, if the `name` property coming from the database is a string, `record.get("name").asString()` will yield the property value as a `String` object.

For more information, see Data types and mapping to Cypher types.

Session configuration

Database selection

It is recommended to **always specify the database explicitly** through the `.withDatabase("<dbName>")` method, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database. If no database is given, the default database set in the Neo4j instance settings is used.

```
// import org.neo4j.driver.SessionConfig;

var session = driver.session(SessionConfig.builder()
    .withDatabase("neo4j").build());
```

Specifying the database through the configuration method is preferred over the `USE` Cypher clause. If the server runs on a cluster, queries with `USE` require server-side routing to be enabled. Queries may also take longer to execute as they may not reach the right cluster member at the first attempt, and need to be routed to one containing the requested database.

Request routing

In a cluster environment, all sessions are opened in write mode, routing them to the leader. You can change this by calling the method `.withRouting(RoutingControl.READ)`. Note that `.executeRead()` and `.executeWrite()` automatically override the session's default access mode.

```
// import org.neo4j.driver.SessionConfig;
// import org.neo4j.driver.AccessMode;

var session = driver.session(SessionConfig.builder()
    .withDatabase("neo4j")
    .withDefaultAccessMode(AccessMode.READ)
    .build());
```

Although executing a *write* query in read mode likely results in a runtime error, **you should not rely on this for access control.** The difference between the two modes is that *read* transactions are routed to any node of a cluster, whereas *write* ones are directed to the leader. In other words, there is no guarantee that a write query submitted in read mode will be rejected.

Similar remarks hold for the `.executeRead()` and `.executeWrite()` methods.

Run queries as a different user (impersonation)

You can execute a query under the security context of a different user with the method `.withImpersonatedUser("<username>")`, specifying the name of the user to impersonate. For this to work, the user under which the `Driver` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `Driver` object.

```
// import org.neo4j.driver.SessionConfig;
// import org.neo4j.driver.RoutingControl;

var session = driver.session(SessionConfig.builder()
    .withDatabase("neo4j")
    .withImpersonatedUser("somebodyElse")
    .build());
```

When impersonating a user, the query is run within the complete security context of the impersonated user and not the authenticated user (i.e. home database, permissions, etc.).

Transaction configuration

You can exert further control on transactions by providing a `TransactionConfig` object as (optional) second parameter to `.executeRead()`, `.executeWrite()`, and `.beginTransaction()`. Use it to specify:

- A transaction timeout. Transactions that run longer will be terminated by the server. The default value is set on the server side. The minimum value is one millisecond.
- A map of metadata that gets attached to the transaction. These metadata get logged in the server `query.log`, and are visible in the output of the `SHOW TRANSACTIONS` Cypher command. Use this to tag transactions.

```
// import java.time.Duration
// import org.neo4j.driver.SessionConfig
// import org.neo4j.driver.TransactionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    var people = session.executeRead(tx -> {
        var result = tx.run("MATCH (p:Person) RETURN p");
        return result.list();  // return a list of Record objects
    }, TransactionConfig.builder()
        .withTimeout(Duration.ofSeconds(5))
        .withMetadata(Map.of("appName", "peopleTracker"))
        .build()
    );
    people.forEach(person -> System.out.println(person));
}
```

Close sessions

Each connection pool has **a finite number of sessions**, so if you open sessions without ever closing them, your application could run out of them. It is thus recommended to create sessions using the `try-with-resources` statement, which automatically closes them when the application is done with them. When a session is closed, it is returned to the connection pool to be later reused.

If you do not open sessions as resources with `try`, remember to call the `.close()` method when you have finished using them.

```
var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build());

// session usage

session.close();
```

Explore the query execution summary

After all results coming from a query have been processed, the server ends the transaction by returning a summary of execution. It comes as a `ResultSummary` object, and it contains information among which:

- Query counters — What changes the query triggered on the server
- Query execution plan — How the database would execute (or executed) the query
- Notifications — Extra information raised by the server while running the query
- Timing information and query request summary

Retrieve the execution summary

When running queries with `Driver.executableQuery()`, the execution summary is part of the default return object, retrievable through the `.summary()` method.

```
var result = driver.executableQuery("""
    UNWIND ['Alice', 'Bob'] AS name
    MERGE (p:Person {name: name})
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var resultSummary = result.summary();
```

If you are using transaction functions, you can retrieve the query execution summary with the method `Result.consume()`. **Notice that once you ask for the execution summary, the result stream is exhausted.** This means that any record which has not yet been processed is discarded.

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    var resultSummary = session.executeWrite(tx -> {
        var result = tx.run("""
        UNWIND ['Alice', 'Bob'] AS name
        MERGE (p:Person {name: name})
        """);
        return result.consume();
    });
}
```

Query counters

The method `ResultSummary.counters()` returns counters for the operations that a query triggered (as a `SummaryCounters` object).

Insert some data and display the query counters

```
var result = driver.executableQuery("""
    MERGE (p:Person {name: $name})
    MERGE (p)-[:KNOWS]->(:Person {name: $friend})
    """).withParameters(Map.of("name", "Mark", "friend", "Bob"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var queryCounters = result.summary().counters();
System.out.println(queryCounters);

/*
InternalSummaryCounters{nodesCreated=2, nodesDeleted=0, relationshipsCreated=1, relationshipsDeleted=0,
propertiesSet=2, labelsAdded=2, labelsRemoved=0, indexesAdded=0, indexesRemoved=0, constraintsAdded=0,
constraintsRemoved=0, systemUpdates=0}
*/
```

There are two additional boolean methods which act as meta-counters:

- `.containsUpdates()` — Whether the query triggered any write operation on the database on which it ran
- `.containsSystemUpdates()` — Whether the query updated the `system` database

Query execution plan

If you prefix a query with `EXPLAIN`, the server will return the plan it *would* use to run the query, but will not actually run it. The plan is then available as a `Plan` object through the method `ResultSummary.plan()`, and contains the list of Cypher operators that would be used to retrieve the result set. You may use this information to locate potential bottlenecks or room for performance improvements (for example through the creation of indexes).

```
var result = driver.executableQuery("EXPLAIN MATCH (p {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var queryPlan = result.summary().plan().arguments().get("string-representation");
System.out.println(queryPlan);

/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+---------------------+
| Operator        | Details        | Estimated Rows | Pipeline            |
+-----------------+----------------+----------------+---------------------+
| +ProduceResults | p              |              1 |                     |
| |               +----------------+----------------+                     |
| +Filter         | p.name = $name |              1 |                     |
| |               +----------------+----------------+                     |
| +AllNodesScan   | p              |             10 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+---------------------+

Total database accesses: ?
*/
```

If you instead prefix a query with the keyword `PROFILE`, the server will return the execution plan it has used to run the query, together with profiler statistics. This includes the list of operators that were used and additional profiling information about each intermediate step. The plan is available as a `Plan` object through the method `ResultSummary.profile()`. Notice that the query is also *run*, so the result object also contains any result records.

```
var result = driver.executableQuery("PROFILE MATCH (p {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var queryPlan = result.summary().profile().arguments().get("string-representation");
System.out.println(queryPlan);

/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| Operator        | Details        | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache
Hits/Misses | Time (ms) | Pipeline            |
+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| +ProduceResults | p              |              1 |    1 |       3 |                |
|            |                      |
| |               +----------------+----------------+------+---------+----------------+
|            |                      |
| +Filter         | p.name = $name |              1 |    1 |       4 |                |
|            |                      |
| |               +----------------+----------------+------+---------+----------------+
|            |                      |
| +AllNodesScan   | p              |             10 |    4 |       5 |            120 |
9160/0 |   108.923 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+

Total database accesses: 12, total allocated memory: 184
*/
```

For more information and examples, see Basic query tuning.

Notifications

After executing a query, the server can return notifications alongside the query result. Notifications contain recommendations for performance improvements, warnings about the usage of deprecated features, and other hints about sub-optimal usage of Neo4j.

For driver version >= 5.25 and server version >= 5.23, two forms of notifications are available (*Neo4j status codes* and *GQL status codes*). For earlier versions, only *Neo4j status codes* are available.
GQL status codes are planned to supersede Neo4j status codes.

Example 1. An unbounded shortest path raises a performance notification

Filter notifications

By default, the server analyses each query for all categories and severity of notifications. Starting from version 5.22, you can use the configuration methods `.withMinimumNotificationSeverity()` and `.withDisabledNotificationClassification()` to tweak the severity and/or category/classification of notifications that you are interested into, or to disable them altogether. There is a slight performance gain in restricting the amount of notifications the server is allowed to raise.

The severity filter applies to both Neo4j and GQL notifications. The category filter acts on both categories and classifications.

You can call the methods both on a `Config` object when creating a `Driver` instance, and on a `SessionConfig` object when creating a session.

Allow only `WARNING` notifications, but not of `HINT` or `GENERIC` classifications

```
// import java.util.Set
// import org.neo4j.driver.Config;
// import org.neo4j.driver.NotificationClassification;
// import org.neo4j.driver.NotificationConfig;
// import org.neo4j.driver.NotificationSeverity;
// import org.neo4j.driver.SessionConfig;

// at `Driver` level
var driver = GraphDatabase.driver(
    dbUri, AuthTokens.basic(dbUser, dbPassword),
    Config.builder()
    .withMinimumNotificationSeverity(NotificationSeverity.WARNING)  // NotificationSeverity.OFF to disable
entirely
    .withDisabledNotificationClassifications(Set.of(NotificationClassification.PERFORMANCE,
NotificationClassification.GENERIC))  // filters categories as well
    .build()
);

// at `Session` level
var session = driver.session(
    SessionConfig.builder()
    .withDatabase("neo4j")
    .withMinimumNotificationSeverity(NotificationSeverity.WARNING)  // NotificationSeverity.OFF to disable
entirely
    .withDisabledNotificationClassifications(Set.of(NotificationClassification.PERFORMANCE,
NotificationClassification.GENERIC))  // filters categories as well
    .build()
);
```

Notifications filtering on versions earlier than 5.22

For versions earlier than 5.22, notification filtering is done via the configuration method `.withNotificationConfig()` (versions 5.7+).

The `NotificationConfig` interface provides the methods `.enableMinimumSeverity()`, `.disableCategories()`, and `.disableAllConfig()` to set the configuration.

Allow only `WARNING` notifications, but not of `HINT` or `GENERIC` category

```
// import java.util.Set
// import org.neo4j.driver.Config;
// import org.neo4j.driver.NotificationCategory;
// import org.neo4j.driver.NotificationConfig;
// import org.neo4j.driver.NotificationSeverity;
// import org.neo4j.driver.SessionConfig;

// at `Driver` level
var driver = GraphDatabase.driver(
    dbUri, AuthTokens.basic(dbUser, dbPassword),
    Config.builder()
    .withNotificationConfig(NotificationConfig.defaultConfig()
        .enableMinimumSeverity(NotificationSeverity.WARNING)
        .disableCategories(Set.of(NotificationCategory.HINT, NotificationCategory.GENERIC))
    ).build()
);

// at `Session` level
var session = driver.session(
    SessionConfig.builder()
    .withDatabase("neo4j")
    .withNotificationConfig(NotificationConfig.defaultConfig()
        .enableMinimumSeverity(NotificationSeverity.WARNING)
        .disableCategories(Set.of(NotificationCategory.HINT, NotificationCategory.GENERIC))
    ).build()
);
```

Disable all notifications

```
// import org.neo4j.driver.Config;
// import org.neo4j.driver.NotificationConfig;
// import org.neo4j.driver.SessionConfig;

// at `Driver` level
var driver = GraphDatabase.driver(
    dbUri, AuthTokens.basic(dbUser, dbPassword),
    Config.builder()
    .withNotificationConfig(NotificationConfig.disableAllConfig())
    .build()
);

// at `Session` level
var session = driver.session(
    SessionConfig.builder()
    .withDatabase("neo4j")
    .withNotificationConfig(NotificationConfig.disableAllConfig())
    .build()
);
```

Run non-blocking asynchronous queries

The examples in Query the database and Run your own transactions use the driver in its synchronous form. This means that, when running a query against the database, your application waits for the server to retrieve all the results and transmit them to the driver. This is not a problem for most use cases, but for queries that have a long processing time or a large result set, asynchronous handling may speed up your application.

Asynchronous managed transactions

You run an asynchronous transaction through an `AsyncSession`. The flow is similar to that of regular transactions, except that async transaction functions return a `CompletionStage` object (which may be further converted into `CompletableFuture`).

```
package demo;

import java.util.Map;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

import org.neo4j.driver.async.AsyncSession;
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.summary.ResultSummary;
import org.neo4j.driver.SessionConfig;

public class App {

    public static void main(String... args) throws ExecutionException, InterruptedException {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {  ①
            driver.verifyConnectivity();

            var summary = printAllPeople(driver);
            // Block as long as necessary (for demonstration purposes)
            System.out.println(summary.get());  ⑧
        }
    }

    public static CompletableFuture<ResultSummary> printAllPeople(Driver driver) {
        var session = driver.session(AsyncSession.class, SessionConfig.builder().withDatabase("neo4j"
).build());  ②

        var query = """
        UNWIND ['Alice', 'Bob', 'Sofia', 'Charles'] AS name
        MERGE (p:Person {name: name}) RETURN p.name
        """;
        var summary = session.executeWriteAsync(tx -> tx.runAsync(query)  ③
                .thenCompose(resCursor -> resCursor.forEachAsync(record -> {  ④
                    System.out.println(record.get(0).asString());
                })))
            .whenComplete((result, error) -> {  ⑤
                session.closeAsync();
            })
            .toCompletableFuture();  ⑥

        return summary;  ⑦
    }
}
```

① Driver creation is the same in the synchronous and asynchronous versions.

② An asynchronous session is created by providing `AsyncSession.class` as first parameter to `Driver.session()`, which returns an `AsyncSession` object. Note that async sessions may not be opened as resources with `try` statements, as the driver can't know when it is safe to close them.

③ As for regular transactions, `.executeWriteAsync()` (and `executeReadAsync()`) take a transaction function callback. Inside the transaction function, run queries with `.runAsync()`. Each query run returns a `CompletionStage`.

④ Optionally use methods on `CompletionStage` to process the result in the asynchronous runner. The query's result set is available as an `AsyncResultCursor`, which implements a similar set of methods for processing the result to those of synchronous transactions (see Transactions → Process query results). Inner object types are the same as the synchronous case (i.e. `Record`, `ResultSummary`).

⑤ Optionally run operations once the query has completed, such as closing the driver session.

⑥ `CompletableFuture` is a convenient type to return back to the caller.

⑦ Contrary to synchronous transactions, `.executeWriteAsync()` and `executeReadAsync()` return the result summary only. **Result processing and handling must be done inside the asynchronous runner.**

⑧ `.get()` waits as long as necessary for the future to complete, and then returns its result.

The methods `.executeReadAsync()` and `.executeWriteAsync()` have replaced `.readTransactionAsync()` and `.writeTransactionAsync()`, which are deprecated in version 5.x and will be removed in version 6.0.

Coordinate parallel transactions

When working with a Neo4j cluster, causal consistency is enforced by default in most cases, which guarantees that a query is able to read changes made by previous queries. The same does not happen by default for multiple transactions running in parallel though. In that case, you can use *bookmarks* to have one transaction wait for the result of another to be propagated across the cluster before running its own work. This is not a requirement, and **you should only use bookmarks if you *need* casual consistency across different transactions**, as waiting for bookmarks can have a negative performance impact.

A *bookmark* is a token that represents some state of the database. By passing one or multiple bookmarks along with a query, the server will make sure that the query does not get executed before the represented state(s) have been established.

Bookmarks with `.executableQuery()`

When querying the database with `.executableQuery()`, the driver manages bookmarks for you. In this case, you have the guarantee that subsequent queries can read previous changes with no further action.

```
driver.executableQuery("<QUERY 1>").execute();

// subsequent .executableQuery() calls will be causally chained

driver.executableQuery("<QUERY 2>").execute();  // can read result of <QUERY 1>
driver.executableQuery("<QUERY 3>").execute();  // can read result of <QUERY 2>
```

To disable bookmark management and causal consistency, use `.withBookmarkManager(null)` in the query configuration.

```
driver.executableQuery("<QUERY>")
    .withConfig(QueryConfig.builder().withBookmarkManager(null).build())
    .execute();
```

Bookmarks within a single session

Bookmark management happens automatically for queries run within a single session, so that you can trust that queries inside one session are causally chained.

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.executeWriteWithoutResult(tx -> tx.run("<QUERY 1>"));
    session.executeWriteWithoutResult(tx -> tx.run("<QUERY 2>"));  // can read QUERY 1
    session.executeWriteWithoutResult(tx -> tx.run("<QUERY 3>"));  // can read QUERY 1,2
}
```

Bookmarks across multiple sessions

If your application uses multiple sessions, you may need to ensure that one session has completed all its transactions before another session is allowed to run its queries.

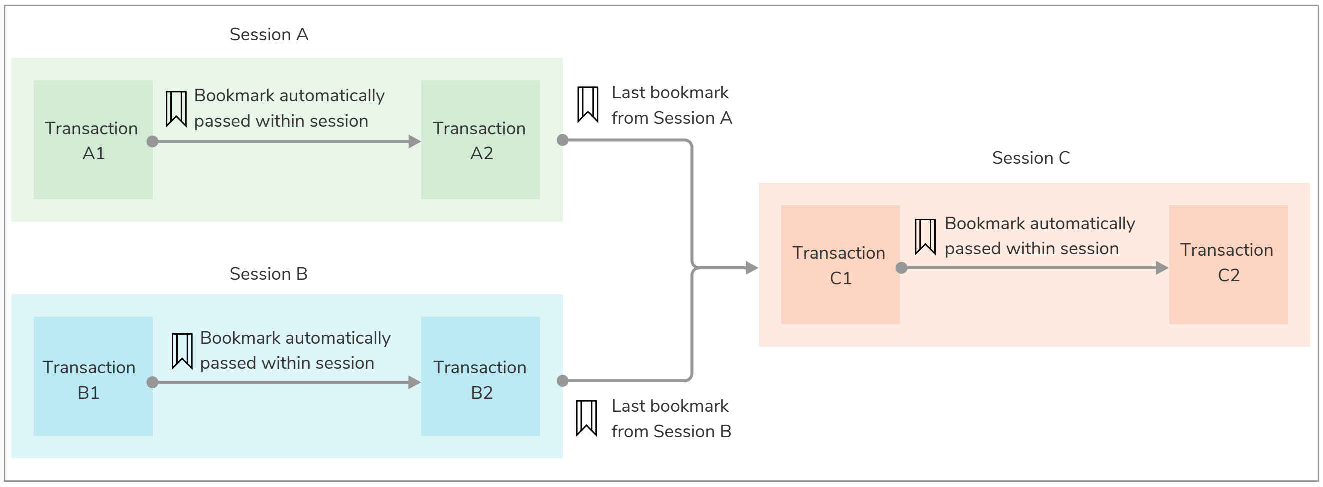
In the example below, `sessionA` and `sessionB` are allowed to run concurrently, while `sessionC` waits until their results have been propagated. This guarantees the `Person` nodes `sessionC` wants to act on actually exist.

Coordinate multiple sessions using bookmarks

```java
package demo;

import java.util.Map;
import java.util.List;
import java.util.ArrayList;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Bookmark;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.SessionConfig;
import org.neo4j.driver.TransactionContext;

public class App {

    private static final int employeeThreshold = 10;

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            createSomeFriends(driver);
        }
    }

    public static void createSomeFriends(Driver driver) {
        List<Bookmark> savedBookmarks = new ArrayList<>();  // to collect the sessions' bookmarks

        // Create the first person and employment relationship
        try (var sessionA = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
            sessionA.executeWriteWithoutResult(tx -> createPerson(tx, "Alice"));
            sessionA.executeWriteWithoutResult(tx -> employ(tx, "Alice", "Wayne Enterprises"));
            savedBookmarks.addAll(sessionA.lastBookmarks());  ①
        }

        // Create the second person and employment relationship
        try (var sessionB = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
            sessionB.executeWriteWithoutResult(tx -> createPerson(tx, "Bob"));
            sessionB.executeWriteWithoutResult(tx -> employ(tx, "Bob", "LexCorp"));
            savedBookmarks.addAll(sessionB.lastBookmarks());  ①
        }

        // Create a friendship between the two people created above
        try (var sessionC = driver.session(SessionConfig.builder()
            .withDatabase("neo4j")
            .withBookmarks(savedBookmarks)  ②
            .build())) {
            sessionC.executeWriteWithoutResult(tx -> createFriendship(tx, "Alice", "Bob"));
            sessionC.executeWriteWithoutResult(tx -> printFriendships(tx));
        }
    }

    // Create a person node
    static void createPerson(TransactionContext tx, String name) {
        tx.run("MERGE (:Person {name: $name})", Map.of("name", name));
    }

    // Create an employment relationship to a pre-existing company node
    // This relies on the person first having been created.
    static void employ(TransactionContext tx, String personName, String companyName) {
        tx.run("""
            MATCH (person:Person {name: $personName})
            MATCH (company:Company {name: $companyName})
            CREATE (person)-[:WORKS_FOR]->(company)
            """, Map.of("personName", personName, "companyName", companyName)
        );
    }
    // Create a friendship between two people
    static void createFriendship(TransactionContext tx, String nameA, String nameB) {
        tx.run("""
            MATCH (a:Person {name: $nameA})
            MATCH (b:Person {name: $nameB})
            MERGE (a)-[:KNOWS]->(b)
            """, Map.of("nameA", nameA, "nameB", nameB)
        );
    }

    // Retrieve and display all friendships
    static void printFriendships(TransactionContext tx) {
        var result = tx.run("MATCH (a)-[:KNOWS]->(b) RETURN a.name, b.name");
        while (result.hasNext()) {
            var record = result.next();
            System.out.println(record.get("a.name").asString() + " knows " + record.get("b.name").
asString());
        }
    }
}
```

① Collect and combine bookmarks from different sessions using `Session.lastBookmarks()`, storing them in a `Bookmark` object.

② Use them to initialize another session with the `.withBookmarks()` config method.


Session A
Transaction A1
Bookmark automatically passed within session
Transaction A2
Last bookmark from Session A
Session B
Transaction B1
Bookmark automatically passed within session
Transaction B2
Last bookmark from Session B
Session C
Transaction C1
Bookmark automatically passed within session
Transaction C2

The use of bookmarks can negatively impact performance, since all queries are forced to wait for the latest changes to be propagated across the cluster. For simple use-cases, try to group queries within a single transaction, or within a single session.

Mix `.executableQuery()` and sessions

To ensure causal consistency among transactions executed partly with `.executableQuery()` and partly with sessions, you can retrieve the default bookmark manager for `ExecutableQuery` instances through `driver.executableQueryBookmarkManager()` and pass it to new sessions through the `.withBookmarkManager()` config method. This will ensure that all work is executed under the same bookmark manager and thus causally consistent.

```java
// import org.neo4j.driver.Driver;
// import org.neo4j.driver.SessionConfig;

driver.executableQuery("<QUERY 1>").execute();

try (var session = driver.session(SessionConfig.builder()
    .withBookmarkManager(driver.executableQueryBookmarkManager())
    .build())) {

    // every query inside this session will be causally chained
    // (i.e., can read what was written by <QUERY 1>)
    session.executeWriteWithoutResult(tx -> tx.run("<QUERY 2>"));
}

// subsequent executableQuery calls will also be causally chained
// (i.e., can read what was written by <QUERY 2>)
driver.executableQuery("<QUERY 3>").execute();
```

Further query mechanisms

Implicit (or auto-commit) transactions

This is the most basic and limited form with which to run a Cypher query. The driver will not automatically retry implicit transactions, as it does instead for queries run with `.executableQuery()` and with managed transactions. Implicit transactions should only be used when the other driver query interfaces do not fit the purpose, or for quick prototyping.

You run an implicit transaction with the method `Session.run()`. It returns a `Result` object that needs to be processed accordingly.

```
// import java.util.Map
// import org.neo4j.driver.SessionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.run("CREATE (a:Person {name: $name})", Map.of("name", "Licia"));
}
```

An implicit transaction gets committed *at the latest* when the session is destroyed, or before another transaction is executed within the same session. Other than that, there is no clear guarantee on when exactly an implicit transaction will be committed during the lifetime of a session. To ensure an implicit transaction is committed, you can call the `.consume()` method on its result.

Since the driver cannot figure out whether the query in a `Session.run()` call requires a read or write session with the database, it defaults to write. If your implicit transaction contains read queries only, there is a performance gain in making the driver aware through the config method `.withRouting(RoutingControl.READ)` when creating the session.

Implicit transactions are the only ones that can be used for **`CALL { … } IN TRANSACTIONS`** queries.

Import CSV files

The most common use case for using `Session.run()` is for importing large CSV files into the database with the `LOAD CSV` Cypher clause, and preventing timeout errors due to the size of the transaction.

Import CSV data into a Neo4j database

```
// import java.util.Map
// import org.neo4j.driver.SessionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    var result = session.run("""
        LOAD CSV FROM 'https://data.neo4j.com/bands/artists.csv' AS line
        CALL {
            WITH line
            MERGE (:Artist {name: line[1], age: toInteger(line[2])})
        } IN TRANSACTIONS OF 2 ROWS
    """);
    var summary = result.consume();
    System.out.println(summary.counters());
}
```

While `LOAD CSV` can be a convenience, there is nothing *wrong* in deferring the parsing of the CSV file to your Java application and avoiding `LOAD CSV`. In fact, moving the parsing logic to the application can give you more control over the importing process. For efficient bulk data insertion, see Performance → Batch data creation.

For more information, see Cypher → Clauses → LOAD CSV.

Transaction configuration

You can exert further control on implicit transactions by providing a `TransactionConfig` object as optional last parameter to `Session.run()` calls. The configuration callbacks allow to specify a query timeout and to attach metadata to the transaction. For more information, see Transactions → Transaction configuration.

```
// import java.util.Map
// import java.time.Duration
// import org.neo4j.driver.SessionConfig
// import org.neo4j.driver.TransactionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    var result = session.run("CREATE (a:Person {name: $name})", Map.of("name", "John"),
        TransactionConfig.builder()
            .withTimeout(Duration.ofSeconds(5))
            .withMetadata(Map.of("appName", "peopleTracker"))
            .build()
    );
}
```

Dynamic values in property keys, relationship types, and labels

In general, you should not concatenate parameters directly into a query, but rather use query parameters. There can however be circumstances where your query structure prevents the usage of parameters in all its parts. In fact, although parameters can be used for literals and expressions as well as node and relationship ids, they cannot be used for the following constructs:

- property keys, so `MATCH (n) WHERE n.$param = 'something'` is invalid;
- relationship types, so `MATCH (n)-[:$param]→(m)` is invalid;
- labels, so `MATCH (n:$param)` is invalid.

For those queries, you are forced to use string concatenation. To protect against Cypher injections, you should enclose the dynamic values in backticks and escape them yourself. Notice that Cypher processes Unicode, so take care of the Unicode literal `\u0060` as well.

Manually escaping dynamic labels before concatenation.

```
// import org.neo4j.driver.QueryConfig;

var label = "Person\\u0060n";
// convert \u0060 to literal backtick and then escape backticks
var escapedLabel = label.replace("\\u0060", "`").replace("`", "``");

var result = driver.executableQuery("MATCH (p:`" + escapedLabel + "` {name: $name}) RETURN p.name")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

Another workaround, which avoids string concatenation, is using APOC procedures, such as `apoc.merge.node`, which supports dynamic labels and property keys.

Using `apoc.merge.node` to create a node with dynamic labels/property keys.

```
// import org.neo4j.driver.QueryConfig;

String propertyKey = "name";
String label = "Person";

var result = driver.executableQuery("CALL apoc.merge.node($labels, $properties)")
    .withParameters(Map.of("labels", List.of(label), "properties", Map.of(propertyKey, "Alice")))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

If you are running Neo4j in Docker, APOC needs to be enabled when starting the container. See APOC → Installation → Docker.

Control results flow with reactive streams

In a reactive flow, consumers dictate the rate at which they consume records from queries, and the driver in turn manages the rate at which records are requested from the server.

An example use-case is an application fetching records from a Neo4j server and doing some very time-consuming post-processing on each one. If the server were allowed to push records to the client as soon as it has them available, the client may be overflown with a lot of entries while its processing is still lagging behind. The Reactive API ensures that the receiving side is not forced to buffer arbitrary amounts of data.

The driver's reactive implementation lives in the `reactivestreams` sub-package and relies on the `reactor-core` package from Project Reactor.

ℹ	The Reactive API is recommended for applications that already work in a reactive programming style, and which have needs that only Reactive workflows can address. For all other cases, the sync and async APIs are recommended.

Install dependencies

To use reactive features, you need to add the relevant dependencies to your project first (refer to Reactor → Reference → Getting reactor).

1. Add Reactor's BOM to your `pom.xml` in a `dependencyManagement` section. Notice that this is in addition to the regular `dependencies` section. If a `dependencyManagement` section already exists in your pom, add only the contents.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>2023.0.2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. Add the `reactor-core` dependency to the `dependencies` section. Notice that the version tag is omitted (it is picked up from Reactor's BOM).

```xml
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
</dependency>
```

Reactive query examples

The basic driver's concepts are the same as the synchronous case, but queries are run through a `ReactiveSession`, and the objects related to querying have a reactive counterpart and prefix.

Managed transaction with reactive sessions

A managed transaction `.executeRead()` example

```java
package demo;

import java.util.List;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Record;
import org.neo4j.driver.SessionConfig;
import org.neo4j.driver.Value;
import org.neo4j.driver.reactivestreams.ReactiveResult;
import org.neo4j.driver.reactivestreams.ReactiveSession;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            Flux<Record> records = Flux.usingWhen(  ①
                Mono.just(driver.session(  ②
                    ReactiveSession.class,  ③
                    SessionConfig.builder().withDatabase("neo4j").build()
                )),
                rxSession -> Mono.fromDirect(rxSession.executeRead(  ④
                    tx -> Mono
                        .fromDirect(tx.run("UNWIND range (1, 5) AS x RETURN x"))  ⑤
                        .flatMapMany(ReactiveResult::records)  ⑥
                )),
                ReactiveSession::close  ⑦
            );

            // block for demonstration purposes
            List<Value> values = records.map(record -> record.get("x")).collectList().block();  ⑧
            System.out.println(values);
        }
    }
}
```

① `Flux.usingWhen(resourceSupplier, workerClosure, cleanupFunction)` is used to create a new session, run queries using it, and finally close it. It will ensure the resource is alive for the time it is needed, and allows to specify the cleanup operation to undertake at the end.

② `.usingWhen()` takes a *resource supplier* in the form of a `Publisher`, hence why session creation is wrapped in a `Mono.just()` call, which spawns a `Mono` from any value.

③ The session creation is similar to the async case, and the same configuration methods apply. The difference is that the first argument must be `ReactiveSession.class`, and the return value is a `ReactiveSession` object.

④ The method `ReactiveSession.executeRead()` runs a read transaction and returns a `Publisher` with the callee's return, which `Mono.fromDirect()` converts into a `Mono`.

⑤ The method `tx.run()` returns a `Publisher<ReactiveResult>`, which `Mono.fromDirect()` converts into a `Mono`.

⑥ Before the final result is returned, `Mono.flatMapMany()` retrieves the records from the result and returns them as a new `Flux`.

⑦ The final cleanup closes the session.

⑧ To show the result of the reactive workflow, `.block()` waits for the flow to complete so that values can be printed. In a real application you wouldn't block but rather forward the records publisher to your framework of choice, which would process them in a meaningful way.

You may run several queries within the same reactive session through several calls to `executeRead/Write()` within the `workerClosure`.

Implicit transaction with reactive sessions

The following example is very similar to the previous one, except it uses an implicit transaction.

An implicit transaction `.run()` example

```
package demo;

import java.util.List;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Record;
import org.neo4j.driver.SessionConfig;
import org.neo4j.driver.Value;
import org.neo4j.driver.reactivestreams.ReactiveResult;
import org.neo4j.driver.reactivestreams.ReactiveSession;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            Flux<Record> records = Flux.usingWhen(
                Mono.just(driver.session(
                    ReactiveSession.class,
                    SessionConfig.builder().withDatabase("neo4j").build()
                )),
                rxSession -> Mono
                    .fromDirect(rxSession.run("UNWIND range (1, 5) AS x RETURN x"))
                    .flatMapMany(ReactiveResult::records),
                ReactiveSession::close
            );

            // block for demonstration purposes
            List<Value> values = records.map(record -> record.get("x")).collectList().block();
            System.out.println(values);
        }
    }
}
```

Always defer session creation

It's important to remember that in reactive programming **a Publisher doesn't come to life until a**

Subscriber attaches to it. A Publisher is just an abstract description of your asynchronous process, but it's only the act of subscribing that triggers the flow of data in the whole chain.

For this reason, always be mindful to make session creation/destruction part of this chain, and not to create sessions separately from the query Publisher chain. Doing so may result in many open sessions, none doing work and all waiting for a Publisher to use them, potentially exhausting the number of available sessions for your application. The previous examples use `Flux.usingWhen()` to address this.

Bad practice example — session is created but nobody uses it

```
ReactiveSession rxSession = driver.session(ReactiveSession.class);
Mono<ReactiveResult> rxResult = Mono.fromDirect(rxSession.run("UNWIND range (1, 5) AS x RETURN x"));
// until somebody subscribes to `rxResult`, the Publisher doesn't materialize, but the session is busy!
```

Performance recommendations

Always specify the target database

Specify the target database on all queries with the `.withDatabase()` method, either in `Driver.executableQuery()` calls or when creating new sessions. If no database is provided, the driver has to send an extra request to the server to figure out what the default database is. The overhead is minimal for a single query, but becomes significant over hundreds of queries.

Good practices

```
driver.executableQuery("<QUERY>")
    .withConfig(QueryConfig.builder().withDatabase("<DB NAME>").build())
    .execute();
```

```
driver.session(SessionConfig.builder().withDatabase("<DB NAME>").build());
```

Bad practices

```
driver.executableQuery("<QUERY>")
    .execute();
```

```
driver.session();
```

Be aware of the cost of transactions

When submitting queries through `.executableQuery()` or through `.executeRead/Write()`, the server automatically wraps them into a transaction. This behavior ensures that the database always ends up in a consistent state, regardless of what happens during the execution of a transaction (power outages, software crashes, etc).

Creating a safe execution context around a number of queries yields an overhead that is not present if the driver just shoots queries at the server and hopes they will get through. The overhead is small, but can add up as the number of queries increases. For this reason, if your use case values throughput more than data integrity, you may extract further performance by running all queries within a single (auto-commit) transaction. You do this by creating a session and using `session.run()` to run as many queries as needed.

Privilege throughput over data integrity

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    for (int i=0; i<1000; i++) {
        session.run("<QUERY>");
    }
}
```

Privilege data integrity over throughput

```
for (int i=0; i<1000; i++) {
    driver.executableQuery("<QUERY>").execute();
    // or session.executeRead/Write() calls
}
```

Route read queries to cluster readers

In a cluster, **route read queries to** secondary nodes. You do this by:

- using the method `.withRouting(RoutingControl.READ)` in `Driver.executableQuery()` calls
- using `Session.executeRead()` instead of `Session.executeWrite()` (for managed transactions)
- using the method `.withRouting(RoutingControl.READ)` when creating a new session (for explicit transactions).

Good practices

```
// import org.neo4j.driver.RoutingControl;

driver.executableQuery("MATCH (p:Person) RETURN p")
    .withConfig(QueryConfig.builder()
        .withDatabase("neo4j")
        .withRouting(RoutingControl.READ)
        .build())
    .execute();
```

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.executeRead(tx -> {
        var result = tx.run("MATCH (p:Person) RETURN p");
        return result.list();
    });
}
```

Bad practices

```
// defaults to routing = writers
driver.executableQuery("MATCH (p:Person) RETURN p")
    .withConfig(QueryConfig.builder()
        .withDatabase("neo4j")
        .build())
    .execute();
```

```
// don't ask to write on a read-only operation
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.executeWrite(tx -> {
        var result = tx.run("MATCH (p:Person) RETURN p");
        return result.list();
    });
}
```

Create indexes

Create indexes for properties that you often filter against. For example, if you often look up `Person` nodes by the `name` property, it is beneficial to create an index on `Person.name`. You can create indexes with the `CREATE INDEX` Cypher clause, for both nodes and relationships.

Create an index on Person.name

```
driver.executableQuery("CREATE INDEX person_name FOR (n:Person) ON (n.name)").execute();
```

For more information, see Indexes for search performance.

Profile queries

Profile your queries to locate queries whose performance can be improved. You can profile queries by prepending them with `PROFILE`. The server output is available through the `.profile()` method of the `ResultSummary` object.

```
var result = driver.executableQuery("PROFILE MATCH (p {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var queryPlan = result.summary().profile().arguments().get("string-representation");
System.out.println(queryPlan);

/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+---------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| Operator        | Details       | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache
Hits/Misses | Time (ms) | Pipeline            |
+-----------------+---------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+
| +ProduceResults | p             |              1 |    1 |       3 |                |
|           |                     |
| |               +---------------+----------------+------+---------+----------------+
|           |                     |
| +Filter         | p.name = $name |              1 |    1 |       4 |                |
|           |                     |
| |               +---------------+----------------+------+---------+----------------+
|           |                     |
| +AllNodesScan   | p             |             10 |    4 |       5 |            120 |
9160/0 |   108.923 | Fused in Pipeline 0 |
+-----------------+---------------+----------------+------+---------+----------------
+------------------------+-----------+---------------------+

Total database accesses: 12, total allocated memory: 184
*/
```

In case some queries are so slow that you are unable to even run them in reasonable times, you can prepend them with `EXPLAIN` instead of `PROFILE`. This will return the *plan* that the server would use to run the query, but without executing it. The server output is available through the `.plan()` method of the `ResultSummary` object.

```
var result = driver.executableQuery("EXPLAIN MATCH (p {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var queryPlan = result.summary().plan().arguments().get("string-representation");
System.out.println(queryPlan);

/*
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+---------------------+
| Operator        | Details        | Estimated Rows | Pipeline            |
+-----------------+----------------+----------------+---------------------+
| +ProduceResults | p              |              1 |                     |
| |               +----------------+----------------+                     |
| +Filter         | p.name = $name |              1 |                     |
| |               +----------------+----------------+                     |
| +AllNodesScan   | p              |             10 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+---------------------+

Total database accesses: ?
*/
```

Specify node labels

Specify node labels in all queries. This allows the query planner to work much more efficiently, and to leverage indexes where available. To learn how to combine labels, see Cypher → Label expressions.

Good practices

```
driver.executableQuery("MATCH (p:Person|Animal {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.run("MATCH (p:Person|Animal {name: $name}) RETURN p", Map.of("name", "Alice"));
}
```

Bad practices

```
driver.executableQuery("MATCH (p {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.run("MATCH (p {name: $name}) RETURN p", Map.of("name", "Alice"));
}
```

Batch data creation

Batch queries when creating a lot of records using the `UNWIND` Cypher clauses.

Good practice

Submit one single queries with all values inside

```
// Generate a sequence of numbers
int start = 1;
int end = 10000;
List<Map> numbers = new ArrayList<>(end - start + 1);
for (int i=start; i<=end; i++) {
    numbers.add(Map.of("value", i));
}

driver.executableQuery("""
    UNWIND $numbers AS node
    CREATE (:Number {value: node.value})
    """)
    .withParameters(Map.of("numbers", numbers))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

Bad practice

Submit a lot of single queries, one for each value

```
for (int i=1; i<=10000; i++) {
driver.executableQuery("CREATE (:Number {value: $value})")
    .withParameters(Map.of("value", i))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
}
```

The most efficient way of performing a *first import* of large amounts of data into a new database is the `neo4j-admin database import` command.

Use query parameters

Always use query parameters instead of hardcoding or concatenating values into queries. Besides protecting from Cypher injections, this allows to better leverage the database query cache.

Good practices

```
driver.executableQuery("MATCH (p:Person {name: $name}) RETURN p")
    .withParameters(Map.of("name", "Alice"))
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.run("MATCH (p:Person {name: $name}) RETURN p", Map.of("name", "Alice"));
}
```

Bad practices

```
driver.executableQuery("MATCH (p:Person {name: 'Alice'}) RETURN p")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
// or
String name = "Alice";
driver.executableQuery("MATCH (p:Person {name: '" + name + "'}) RETURN p")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

```
try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    session.run("MATCH (p:Person {name: 'Alice'}) RETURN p");
    // or
    String name = "Alice";
    session.run("MATCH (p:Person {name: '" + name + "'}) RETURN p");
}
```

Concurrency

Use asynchronous querying. This is likely to be more impactful on performance if you parallelize complex and time-consuming queries in your application, but not so much if you run many simple ones.

Use `MERGE` for creation only when needed

The Cypher clause `MERGE` is convenient for data creation, as it allows to avoid duplicate data when an exact clone of the given pattern exists. However, it requires the database to run two queries: it first needs to `CREATE` it (if needed).

If you know already that the data you are inserting is new, avoid using `MERGE` and use `CREATE` directly instead — this practically halves the number of database queries.

Filter notifications

Filter the category and/or severity of notifications the server should raise.

=Reference=

Advanced connection information

Connection URI

The driver supports connection to URIs of the form

```
<SCHEME>://<HOST>[:<PORT>[?policy=<POLICY-NAME>]]
```

- `<SCHEME>` is one among `neo4j`, `neo4j+s`, `neo4j+ssc`, `bolt`, `bolt+s`, `bolt+ssc`.
- `<HOST>` is the host name where the Neo4j server is located.
- `<PORT>` is optional, and denotes the port the Bolt protocol is available at.
- `<POLICY-NAME>` is an optional *server policy* name. Server policies need to be set up prior to usage.

The driver does not support connection to a nested path, such as `example.com/neo4j/`. The server must be reachable from the domain root.

Connection protocols and security

Communication between the driver and the server is mediated by Bolt. The scheme of the server URI determines whether the connection is encrypted and, if so, what type of certificates are accepted.

URL scheme	Encryption	Comment
neo4j	✗	Default for local setups
neo4j+s	✓ (only CA-signed certificates)	Default for Aura
neo4j+ssc	✓ (CA- and self-signed certificates)	

The driver receives a *routing table* from the server upon successful connection, regardless of whether the instance is a proper cluster environment or a single-machine environment. The driver's routing behavior works in tandem with Neo4j's clustering by directing read/write transactions to appropriate cluster members. If you want to target a *specific* machine, use the `bolt`, `bolt+s`, or `bolt+ssc` URI schemes instead.

The connection scheme to use is not your choice, but is rather determined by the server requirements. You must know the right server scheme upfront, as no metadata is exposed prior to connection. If you are unsure, ask the database administrator.

Authentication methods

Basic authentication (default)

The basic authentication scheme relies on traditional username and password. These can either be the credentials for your local installation, or the ones provided with an Aura instance.

```
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.GraphDatabase;

GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword));
```

The basic authentication scheme can also be used to authenticate against an LDAP server (Enterprise Edition only).

Kerberos authentication

The Kerberos authentication scheme requires a base64-encoded ticket. It can only be used if the server has the Kerberos Add-on installed.

```
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.GraphDatabase;

GraphDatabase.driver(dbUri, AuthTokens.kerberos(ticket));
```

Bearer authentication

The bearer authentication scheme requires a base64-encoded token provided by an Identity Provider through Neo4j's Single Sign-On feature.

```
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.GraphDatabase;

GraphDatabase.driver(dbUri, AuthTokens.bearer(ticket));
```

The bearer authentication scheme requires configuring Single Sign-On on the server. Once configured, clients can discover Neo4j's configuration through the Discovery API.

Custom authentication

Use `AuthTokens.custom()` to log into a server having a custom authentication scheme.

```
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.GraphDatabase;

GraphDatabase.driver(dbUri, AuthTokens.custom(principal, credentials, realm, scheme, parameters));
```

No authentication

If authentication is disabled on the server, the authentication parameter can be omitted entirely.

Logging

By default, the driver logs `INFO` messages through the Java logging framework `java.util.logging`. To change the driver's logging behavior, use the `.withLogging()` method when creating a `Driver` object.

```java
// import java.util.logging.Level;
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.Config;
// import org.neo4j.driver.GraphDatabase;
// import org.neo4j.driver.Logging;

try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword),
    Config.builder().withLogging(Logging.console(Level.FINE)).build())) {
    driver.verifyConnectivity();
}
```

Example of log output upon driver connection

```
2023-12-22T10:36:39.997882867 INFO org.neo4j.driver.internal.DriverFactory - Routing driver instance
1651855867 created for server address localhost:7687
2023-12-22T10:36:40.03430944 FINE io.netty.channel.DefaultChannelId - -Dio.netty.processId: 23665 (auto-
detected)
2023-12-22T10:36:40.036871656 FINE io.netty.util.NetUtil - -Djava.net.preferIPv4Stack: false
2023-12-22T10:36:40.037023871 FINE io.netty.util.NetUtil - -Djava.net.preferIPv6Addresses: false
2023-12-22T10:36:40.03827624 FINE io.netty.util.NetUtilInitializations - Loopback interface: lo (lo,
0:0:0:0:0:0:0:1%lo)
2023-12-22T10:36:40.038877108 FINE io.netty.util.NetUtil - /proc/sys/net/core/somaxconn: 4096
2023-12-22T10:36:40.03958947 FINE io.netty.channel.DefaultChannelId - -Dio.netty.machineId:
04:cf:4b:ff:fe:0e:ee:99 (auto-detected)
2023-12-22T10:36:40.04531968 FINE io.netty.util.ResourceLeakDetector - -Dio.netty.leakDetection.level:
simple
2023-12-22T10:36:40.045471749 FINE io.netty.util.ResourceLeakDetector -
-Dio.netty.leakDetection.targetRecords: 4
2023-12-22T10:36:40.059848221 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.numHeapArenas: 40
2023-12-22T10:36:40.060000842 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.numDirectArenas: 40
2023-12-22T10:36:40.060113675 FINE io.netty.buffer.PooledByteBufAllocator - -Dio.netty.allocator.pageSize:
8192
2023-12-22T10:36:40.060219802 FINE io.netty.buffer.PooledByteBufAllocator - -Dio.netty.allocator.maxOrder:
9
2023-12-22T10:36:40.060324679 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.chunkSize: 4194304
2023-12-22T10:36:40.060442554 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.smallCacheSize: 256
2023-12-22T10:36:40.060547232 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.normalCacheSize: 64
2023-12-22T10:36:40.060648929 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.maxCachedBufferCapacity: 32768
2023-12-22T10:36:40.060750268 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.cacheTrimInterval: 8192
2023-12-22T10:36:40.060858214 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.cacheTrimIntervalMillis: 0
2023-12-22T10:36:40.060965492 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.useCacheForAllThreads: false
2023-12-22T10:36:40.061068878 FINE io.netty.buffer.PooledByteBufAllocator -
-Dio.netty.allocator.maxCachedByteBuffersPerChunk: 1023
2023-12-22T10:36:40.069792775 FINE io.netty.buffer.ByteBufUtil - -Dio.netty.allocator.type: pooled
2023-12-22T10:36:40.069957048 FINE io.netty.buffer.ByteBufUtil - -Dio.netty.threadLocalDirectBufferSize: 0
2023-12-22T10:36:40.070070891 FINE io.netty.buffer.ByteBufUtil - -Dio.netty.maxThreadLocalCharBufferSize:
16384
2023-12-22T10:36:40.102235419 FINE io.netty.buffer.AbstractByteBuf - -Dio.netty.buffer.checkAccessible:
true
2023-12-22T10:36:40.102408774 FINE io.netty.buffer.AbstractByteBuf - -Dio.netty.buffer.checkBounds: true
2023-12-22T10:36:40.103026138 FINE io.netty.util.ResourceLeakDetectorFactory - Loaded default
ResourceLeakDetector: io.netty.util.ResourceLeakDetector@1a67b908
2023-12-22T10:36:40.104721387 FINE org.neo4j.driver.internal.async.connection.ChannelConnectedListener -
[0xb354eed2][localhost(127.0.0.1):7687][] C: [Bolt Handshake] [0x6060b017, 263173, 132100, 260, 3]
2023-12-22T10:36:40.106645202 FINE io.netty.util.Recycler - -Dio.netty.recycler.maxCapacityPerThread: 4096
2023-12-22T10:36:40.106785483 FINE io.netty.util.Recycler - -Dio.netty.recycler.ratio: 8
2023-12-22T10:36:40.106887674 FINE io.netty.util.Recycler - -Dio.netty.recycler.chunkSize: 32
2023-12-22T10:36:40.106993748 FINE io.netty.util.Recycler - -Dio.netty.recycler.blocking: false
2023-12-22T10:36:40.107096042 FINE io.netty.util.Recycler - -Dio.netty.recycler.batchFastThreadLocalOnly:
true
2023-12-22T10:36:40.11603651 FINE org.neo4j.driver.internal.async.connection.HandshakeHandler -
[0xb354eed2][localhost(127.0.0.1):7687][] S: [Bolt Handshake] 5.4
2023-12-22T10:36:40.128082306 FINE org.neo4j.driver.internal.async.outbound.OutboundMessageHandler -
[0xb354eed2][localhost(127.0.0.1):7687][] C: HELLO {routing={address: "localhost:7687"},
bolt_agent={product: "neo4j-java/dev", language: "Java/17.0.9", language_details: "Optional[Eclipse
Adoptium; OpenJDK 64-Bit Server VM; 17.0.9+9]", platform: "Linux; 5.15.0-91-generic; amd64"},
user_agent="neo4j-java/dev"}
2023-12-22T10:36:40.130350166 FINE org.neo4j.driver.internal.async.pool.NettyChannelTracker - Channel
[0xb354eed2] created. Local address: /127.0.0.1:32794, remote address: /127.0.0.1:7687
2023-12-22T10:36:40.130746872 FINE org.neo4j.driver.internal.async.pool.NettyChannelTracker - Channel
[0xb354eed2] acquired from the pool. Local address: /127.0.0.1:32794, remote address: /127.0.0.1:7687
2023-12-22T10:36:40.133652153 FINE org.neo4j.driver.internal.async.outbound.OutboundMessageHandler -
[0xb354eed2][localhost(127.0.0.1):7687][] C: LOGON {principal="neo4j", scheme="basic",
credentials="******"}
2023-12-22T10:36:40.140017819 FINE org.neo4j.driver.internal.async.inbound.InboundMessageDispatcher -
[0xb354eed2][localhost(127.0.0.1):7687][] S: SUCCESS {server="Neo4j/5.16.0", connection_id="bolt-5",
hints={connection.recv_timeout_seconds: 120}}
2023-12-22T10:36:40.142229689 FINE org.neo4j.driver.internal.async.inbound.InboundMessageDispatcher -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] S: SUCCESS {}
2023-12-22T10:36:40.14568667 FINE org.neo4j.driver.internal.async.outbound.OutboundMessageHandler -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] C: RESET
2023-12-22T10:36:40.146897982 FINE org.neo4j.driver.internal.async.NetworkConnection - Added
ConnectionReadTimeoutHandler
2023-12-22T10:36:40.14753571 FINE org.neo4j.driver.internal.async.inbound.InboundMessageDispatcher -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] S: SUCCESS {}
2023-12-22T10:36:40.147813446 FINE org.neo4j.driver.internal.async.NetworkConnection - Removed
ConnectionReadTimeoutHandler
2023-12-22T10:36:40.14895232 FINE org.neo4j.driver.internal.async.pool.NettyChannelTracker - Channel
[0xb354eed2] released back to the pool
2023-12-22T10:36:40.15199869 FINE org.neo4j.driver.internal.cluster.RoutingTableRegistryImpl - Routing
table handler for database 'system' is added.
2023-12-22T10:36:40.152613749 FINE org.neo4j.driver.internal.cluster.RoutingTableHandlerImpl - Routing
table for database 'system' is stale. Ttl 1703237800150, currentTime 1703237800152, routers [], writers
[], readers [], database 'system'
2023-12-22T10:36:40.159510973 FINE org.neo4j.driver.internal.async.pool.NettyChannelTracker - Channel
[0xb354eed2] acquired from the pool. Local address: /127.0.0.1:32794, remote address: /127.0.0.1:7687
2023-12-22T10:36:40.165704119 FINE org.neo4j.driver.internal.async.outbound.OutboundMessageHandler -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] C: ROUTE {address="localhost:7687"} [] system null
2023-12-22T10:36:40.168929698 FINE org.neo4j.driver.internal.async.NetworkConnection - Added
ConnectionReadTimeoutHandler
2023-12-22T10:36:40.171700427 FINE org.neo4j.driver.internal.async.inbound.InboundMessageDispatcher -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] S: SUCCESS {rt={servers: [{addresses: ["localhost:7687"],
role: "WRITE"}, {addresses: ["localhost:7687"], role: "READ"}, {addresses: ["localhost:7687"], role:
"ROUTE"}], ttl: 300, db: "system"}}
2023-12-22T10:36:40.17187084 FINE org.neo4j.driver.internal.async.NetworkConnection - Removed
ConnectionReadTimeoutHandler
2023-12-22T10:36:40.173921853 FINE org.neo4j.driver.internal.async.outbound.OutboundMessageHandler -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] C: RESET
2023-12-22T10:36:40.174473474 FINE org.neo4j.driver.internal.async.NetworkConnection - Added
ConnectionReadTimeoutHandler
2023-12-22T10:36:40.175516332 FINE org.neo4j.driver.internal.async.inbound.InboundMessageDispatcher -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] S: SUCCESS {}
2023-12-22T10:36:40.175679271 FINE org.neo4j.driver.internal.async.NetworkConnection - Removed
ConnectionReadTimeoutHandler
2023-12-22T10:36:40.175849144 FINE org.neo4j.driver.internal.async.pool.NettyChannelTracker - Channel
[0xb354eed2] released back to the pool
2023-12-22T10:36:40.182085603 FINE org.neo4j.driver.internal.cluster.RoutingTableHandlerImpl - Fetched
cluster composition for database 'system'. ClusterComposition{readers=[localhost:7687],
writers=[localhost:7687], routers=[localhost:7687], expirationTimestamp=1703238100176,
databaseName=system}
2023-12-22T10:36:40.185015699 FINE org.neo4j.driver.internal.cluster.RoutingTableHandlerImpl - Updated
routing table for database 'system'. Ttl 1703238100176, currentTime 1703237800184, routers
[localhost:7687], writers [localhost:7687], readers [localhost:7687], database 'system'
2023-12-22T10:36:40.18530819 INFO org.neo4j.driver.internal.InternalDriver - Closing driver instance
1651855867
2023-12-22T10:36:40.186508052 FINE org.neo4j.driver.internal.async.outbound.OutboundMessageHandler -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] C: GOODBYE
2023-12-22T10:36:40.187291369 INFO org.neo4j.driver.internal.async.pool.ConnectionPoolImpl - Closing
connection pool towards localhost(127.0.0.1):7687
2023-12-22T10:36:40.189599992 FINE org.neo4j.driver.internal.async.inbound.ChannelErrorHandler -
[0xb354eed2][localhost(127.0.0.1):7687][bolt-5] Channel is inactive
2023-12-22T10:36:40.395356347 FINE io.netty.buffer.PoolThreadCache - Freed 6 thread-local buffer(s) from
thread: Neo4jDriverIO-2-2
```

Custom address resolver

When creating a `Driver` object, you can specify a *resolver* function to resolve the connection address the driver is initialized with. Note that addresses that the driver receives in routing tables are not resolved with the custom resolver.

You specify a resolver through the `.withResolver()` config method, which works with `ServerAddress` objects.

Connection to `example.com` on port `9999` is resolved to `localhost` on port `7687`

```
// import java.util.Set;
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.Config;
// import org.neo4j.driver.GraphDatabase;
// import org.neo4j.driver.net.ServerAddress;

var addresses = Set.of(
    ServerAddress.of("localhost", 7687)  // omit the scheme; provide only host
);
var config = Config.builder()
    .withResolver(address -> addresses)
    .build();
try (var driver = GraphDatabase.driver("neo4j://example.com:9999", AuthTokens.basic(dbUser, dbPassword),
config)) {
    driver.verifyConnectivity();
}
```

OCSP stapling

If OCSP stapling is enabled on the server, the driver can be configured to check for certificate revocations during SSL handshakes. OCSP stapling improves both security and performance when using CA-signed certificates.

There are two methods implementing this feature:

- `.withVerifyIfPresentRevocationChecks()` — validate a certificate's stapling if available, but don't fail verification if no stapling is found.
- `.withStrictRevocationChecks()` — validate a certificate's stapling, and fail verification if no stapling is found.

Both methods act on a `Config.TrustStrategy` object, so you have to be explicit about what certificates you want to trust and cannot rely on the driver to infer it from the connection URI scheme. This means that you have to chain these methods to `Config.TrustStrategy.trustSystemCertificates()`. To avoid configuration clashes, the connection URI scheme must also be set to `neo4j` (i.e. not `neo4j+s` nor `neo4j+ssc`).

Validate certificate stapling, but don't fail if no stapling is found

```
// import org.neo4j.driver.Config;

Config config = Config.builder()
    .withEncryption()
    .withTrustStrategy(Config.TrustStrategy
        .trustSystemCertificates()
        .withVerifyIfPresentRevocationChecks())
    .build();
try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword), config)) {
    driver.verifyConnectivity();
}
```

Further connection parameters

You can find all `Driver` configuration parameters in the API documentation → driver.GraphDatabase.

Data types and mapping to Cypher types

The tables in this section show the mapping between Cypher data types and Java types.

Regardless of their type, all values in query results are embedded within `Value` objects. To extract and cast them to the corresponding Java types, use `.as<type>()` (eg. `.asString()`, `asInt()`, etc). For example, if the `name` property coming from the database is a string, `record.get("name").asString()` will yield the property value as a `String` object.

On the other hand, you don't need to use the driver types when submitting query parameters. The driver will automatically serialize objects of native Java types passed as parameters.

For a complete list of the value types the driver serializes data into, see the API documentation.

Core types

Cypher type	Driver type
NULL	`NullValue`
`LIST`	`ListValue`
`MAP`	`MapValue`
`BOOLEAN`	`BooleanValue`
`INTEGER`	`IntegerValue`
`FLOAT`	`FloatValue`
`STRING`	`StringValue`
`ByteArray`	`BytesValue`

Temporal types

The driver provides a set of temporal data types compliant with ISO-8601 and Cypher. Sub-second values are measured to nanosecond precision.

The driver's types rely on Java's `time` types. All temporal types, except `DurationValue`, are in fact `java.time` objects under the hood. This means that:

- if you want to *query* the database with a temporal type, instantiate a `java.time` object and use it as query parameter (i.e. you don't need to care about driver's types).
- if you *retrieve* a temporal object from the database (including through one of Cypher temporal functions), you will get back the corresponding *driver* type as displayed in the table below. The driver implements methods to convert driver time types into Java ones (ex. `.asZonedDateTime()`, `.asOffsetTime()`, etc).

Cypher type	Driver type
`DATE`	`DateValue`
`ZONED TIME`	`TimeValue`

Cypher type	Driver type
LOCAL TIME	LocalTimeValue
ZONED DATETIME	DateTimeValue
LOCAL DATETIME	LocalDateTimeValue
DURATION	DurationValue

How to use temporal types in queries

```java
package demo;

import java.util.Map;
import java.time.ZonedDateTime;
import java.time.ZoneId;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            // Define a date, with timezone, and use it to set a relationship property
            var friendsSince = ZonedDateTime.of(2016, 12, 16, 13, 59, 59, 9999999, ZoneId.of
("Europe/Stockholm"));
            var result = driver.executableQuery("""
                MERGE (a:Person {name: $name})
                MERGE (b:Person {name: $friend})
                MERGE (a)-[friendship:KNOWS {since: $friendsSince}]->(b)
                RETURN friendship.since AS date
                """)
                .withParameters(Map.of("name", "Alice", "friend", "Bob", "friendsSince", friendsSince))
                .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                .execute();

            var date = result.records().get(0).get("date");
            System.out.println(date.getClass().getName());  //
org.neo4j.driver.internal.value.DateTimeValue
            System.out.println(date);  // 2016-12-16T13:59:59.009999999+01:00[Europe/Stockholm]

            var nativeDate = date.asZonedDateTime();
            System.out.println(nativeDate.getClass().getName());  // java.time.ZonedDateTime
        }
    }
}
```

DurationValue

Represents the difference between two points in time (expressed in months, days, seconds, nanoseconds).

```java
// import org.neo4j.driver.Values;

var duration = Values.isoDuration(1, 2, 3, 4);  // months, days, seconds, nanoseconds
System.out.println(duration);  // P1M2DT3.000000004S
```

For full documentation, see API documentation → DurationValue.

Spatial types

Cypher supports spatial values (points), and Neo4j can store these point values as properties on nodes and relationships.

The attribute `SRID` (short for *Spatial Reference Identifier*) is a number identifying the coordinate system the spatial type is to be interpreted in. You can think of it as a unique identifier for each spatial type.

Cypher type	Driver type	SRID
`POINT` (2D Cartesian)	`PointValue`	7203
`POINT` (2D WGS-84)	`PointValue`	4326
`POINT` (3D Cartesian)	`PointValue`	9157
`POINT` (3D WGS-84)	`PointValue`	4979

You create a point value through `Values.point(srid, x, y[, z])` (the third coordinate is optional). Points returned from database queries are of type `PointValue`, and can be converted to `Point` objects through the method `.asPoint()`.

Receive a `Point` *value from the database*

```java
package demo;

import java.util.Map;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;
import org.neo4j.driver.Values;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            var result = driver.executableQuery("RETURN point({x: 2.3, y: 4.5, z: 2}) AS point")
                .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                .execute();
            var point = result.records().get(0).get("point");
            System.out.println(point);  // Point{srid=9157, x=2.3, y=4.5, z=2.0}
            System.out.println(point.asPoint().x());  // 2.3
        }
    }
}
```

Create a `Point` value and use it as property value

```
package demo;

import java.util.Map;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;
import org.neo4j.driver.Values;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            var location = Values.point(4326, 67.28775180193841, 17.734163823312397);  // 4326 = 2D
geodetic point
            var result = driver.executableQuery("CREATE (p:PlaceOfInterest {location: $location}) RETURN
p")
                .withParameters(Map.of("location", location))
                .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                .execute();
            var place = result.records().get(0).get("p").get("location");
            System.out.println(place);  // Point{srid=4326, x=67.28775180193841, y=17.734163823312397}
            System.out.println(place.asPoint().y());  // 17.734163823312397
        }
    }
}
```

Graph types

Graph types are only passed as results and may not be used as parameters.

Cypher Type	Driver type
`NODE`	`NodeValue`
`RELATIONSHIP`	`RelationshipValue`
`PATH`	`PathValue`

`NodeValue`

Represents a node in a graph.

Table 1. Essential methods on node objects

Method	Return
`.labels()`	Node labels, as a list.
`.asMap()`	Node properties, as a map.
`.get("<propertyName>")`	Value for the given property.

Method	Return
`.elementId()`	String identifier for the relationship. This should be used with care, as no guarantees are given about the mapping between id values and elements outside the scope of a single transaction. In other words, using an `elementId` to `MATCH` an element across different transactions is risky.

Retrieve a node and display its details

```
package demo;

import java.util.Map;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            // Get a node from the database
            var result = driver.executableQuery("MERGE (p:Person:Actor {name: $name, age: 59}) RETURN p")
                .withParameters(Map.of("name", "Alice"))
                .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                .execute();

            // Extract node from result
            var nodeVal = result.records().get(0).get("p");
            var node = nodeVal.asNode();  // .asNode() -> type org.neo4j.driver.types.Node

            System.out.printf("Labels: %s %n", node.labels());
            System.out.printf("Properties: %s %n", node.asMap());
            System.out.printf("Name property: %s %n", node.get("name"));
            System.out.printf("Element ID: %s %n", node.elementId());
            /*
            Labels: [Person, Actor]
            Properties: {name=Alice, age=59}
            Name property: "Alice"
            Element ID: 4:549a0567-2015-4bb6-a40c-8536bf7227b0:5
            */
        }
    }
}
```

For full documentation, see API documentation → NodeValue.

`RelationshipValue`

Represents a relationship in a graph.

Table 2. Essential methods on relationsip objects

Method	Return
`.type()`	Relationship type.
`.asMap()`	Relationship properties, as a map.
`.get("<propertyName>")`	Value for the given property.

Method	Return
`.startNodeElementId()`	`elementId` of starting node.
`.endNodeElementId()`	`elementId` of ending node.
`.elementId()`	String identifier for the relationship. This should be used with care, as no guarantees are given about the mapping between id values and elements outside the scope of a single transaction. In other words, using an `elementId` to `MATCH` an element across different transactions is risky.

Retrieve a relationship and display its details

```
package demo;

import java.util.Map;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.QueryConfig;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            // Get a relationship from the database
            var result = driver.executableQuery("""
                MERGE (p:Person {name: $name})
                MERGE (p)-[r:KNOWS {status: $status, since: date()}]->(friend:Person {name: $friendName})
                RETURN r AS friendship
                """)
                .withParameters(Map.of("name", "Alice", "status", "BFF", "friendName", "Bob"))
                .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                .execute();

            // Extract relationship from result
            var relationshipVal = result.records().get(0).get("friendship");
            var relationship = relationshipVal.asRelationship();  // .asRelationship() -> type
org.neo4j.driver.types.Relationship

            System.out.printf("Type: %s %n", relationship.type());
            System.out.printf("Properties: %s %n", relationship.asMap());
            System.out.printf("Status property: %s %n", relationship.get("status"));
            System.out.printf("Start node: %s %n", relationship.startNodeElementId());
            System.out.printf("End node: %s %n", relationship.endNodeElementId());
            System.out.printf("Element ID: %s %n", relationship.elementId());
            /*
            Type: KNOWS
            Properties: {since=2024-01-12, status=BFF}
            Status property: "BFF"
            Start node: 4:549a0567-2015-4bb6-a40c-8536bf7227b0:0
            End node: 4:549a0567-2015-4bb6-a40c-8536bf7227b0:6
            Element ID: 5:549a0567-2015-4bb6-a40c-8536bf7227b0:1
            */
        }
    }
}
```

For full documentation, see API documentation → RelationshipValue.

`PathValue`

Represents a path in a graph.

The driver breaks paths into (iterable) *segments*, consisting of a start node, one relationship, and an end node. Segments entities may be retrieved, in order, via the methods `.start()`, `.relationship()`, and `.end()`.

Retrieve a path and walk it, listing nodes and relationship

```java
package demo;

import java.util.Map;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.types.Path;
import org.neo4j.driver.QueryConfig;

public class App {

    public static void main(String... args) {
        final String dbUri = "<URI for Neo4j database>";
        final String dbUser = "<Username>";
        final String dbPassword = "<Password>";

        try (var driver = GraphDatabase.driver(dbUri, AuthTokens.basic(dbUser, dbPassword))) {
            driver.verifyConnectivity();

            // Create some :Person nodes linked by :KNOWS relationships
            addFriend(driver, "Alice", "BFF", "Bob");
            addFriend(driver, "Bob", "Fiends", "Sofia");
            addFriend(driver, "Sofia", "Acquaintances", "Sofia");

            // Follow :KNOWS relationships outgoing from Alice three times, return as path
            var result = driver.executableQuery("""
                MATCH path=(:Person {name: $name})-[:KNOWS*3]->(:Person)
                RETURN path AS friendshipChain
                """)
                .withParameters(Map.of("name", "Alice"))
                .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
                .execute();

            // Extract path from result
            var pathVal = result.records().get(0).get("friendshipChain");
            var path = pathVal.asPath();  // .asPath() -> type org.neo4j.driver.types.Path

            System.out.println("-- Path breakdown --");
            for (Path.Segment segment : path) {
                System.out.printf(
                    "%s is friends with %s (%s).%n",
                    segment.start().get("name").asString(),
                    segment.end().get("name").asString(),
                    segment.relationship().get("status").asString());
            }
            /*
            -- Path breakdown --
            Alice is friends with Bob (BFF).
            Bob is friends with Sofia (Fiends).
            Sofia is friends with Sofia (Acquaintances).
            */
        }
    }

    public static void addFriend(Driver driver, String name, String status, String friendName) {
        driver.executableQuery("""
            MERGE (p:Person {name: $name})
            MERGE (p)-[r:KNOWS {status: $status, since: date()}]->(friend:Person {name: $friendName})
            """)
            .withParameters(Map.of("name", name, "status", status, "friendName", friendName))
            .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
            .execute();
    }
}
```

For full documentation, see API documentation → PathValue.

Exceptions

The driver can raise a number of different exceptions, see API documentation → Exceptions. For a list of errors the server can return, see Status codes.

Some server errors are marked as safe to retry without need to alter the original request. Examples of such errors are deadlocks, memory issues, or connectivity issues. Driver’s exceptions implementing `RetryableException` are such that a further attempt at the operation that caused it might be successful. This is particular useful when running queries in explicit transactions, to know if a failed query is worth re-running.

API documentation

Related projects

True to its name (the `j` stands for `java`), Neo4j has a wide Java-ecosystem surrounding it. This page lists the other officially supported Java projects that help you work with a Neo4j database.

- Neo4j OGM — An Object Graph Mapping (OGM) library abstracts the database and provides a convenient way to query it without having to use low level drivers directly.
- Spring Data Neo4j (SDN) — An Object Graph Mapping (OGM) library, as a Spring Data module.
- Spring Boot integration — Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` "Starter".
- Quarkus Neo4j — The Quarkus Neo4j extension provides an instance of the Neo4j driver configured for usage in a Quarkus application.
- Neo4j-Migrations — A database migration and refactoring tool that allows running Cypher scripts and programmatic refactorings in a controlled and repeatable fashion against one or more Neo4j databases.
- Cypher-DSL — A Cypher generator, to dynamically create Cypher queries without doing string concatenation.

=GraphAcademy courses=

Graph Data Modeling Fundamentals

Intermediate Cypher Queries

Building Neo4j Applications with Java

License