Advanced Java Programming Course

Neo4j for Java Developers

Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

- Install the Neo4j Java Driver.
- Build a Connection String.
- Create an instance of the Driver.
- Verify that the Driver has successfully connected to Neo4j.
- Interacting with Neo4j
  - Open a Session and execute a Unit of Work within a Transaction.
  - Execute Read and Write queries through the Driver.
  - Consume the results returned from Neo4j.
  - Handle potential errors thrown by the Driver.

Neo4j Java Driver

- Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.
  - The Driver object is a thread-safe, application-wide fixture from which all Neo4j interaction derives.
  - The Driver API is topology independent, so you can run the same code against a Neo4j cluster or a single DBMS.
- To connect to and query Neo4j from within a Java application, you use the Neo4j Java Driver.

Neo4j Java Driver

- The Neo4j Java Driver is one of five officially supported drivers, the others are JavaScript, .NET, Python, and Go.
- There are also a wide range of Community Drivers available for other languages including PHP and Ruby.
- You should create a single instance of the Driver in your application per Neo4j cluster or DBMS, which can then be shared across your application.

Installing the Driver

- Using Maven

```
<dependency>
    <groupId>org.neo4j.driver</groupId>
    <artifactId>neo4j-java-driver</artifactId>
    <version>5.17.0</version>
</dependency>
```

- Using Gradle

```
implementation 'org.neo4j.driver:neo4j-java-driver:5.17.0'
```

Creating a Driver Instance

- Each driver instance will connect to one DBMS, or Neo4j cluster, depending on the value provided in the connection string.
- All in org.neo4j.driver.* package
- Using GraphDatabase.driver() factory method call to create a driver instance.

Creating a Driver Instance

- The driver() method accepts the following arguments:
  - A connection string for the Neo4j cluster or DBMS
    - neo4j://localhost:7687
    - neo4j+s://dbhash.databases.neo4j.io:7687
  - An authentication token
    - Neo4j supports basic username and password authentication
    - Can use the static methods provided by AuthTokens class to create an authentication token.
    - AuthToken authToken = AuthTokens.basic(username, password);
  - Optionally, an object containing additional driver configuration

```
Config config = Config.builder()
        .withConnectionTimeout(30, TimeUnit.SECONDS)
        .withMaxConnectionPoolSize(10)
        .build();
```

Creating a Driver Instance

- Connect to the database

```
import org.neo4j.driver.*;


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();
        }
    }
}
```

Creating a Driver Instance

- An example for how to create a driver instance

```
import org.neo4j.driver.*;

public class AppUtils {

        public static Driver initDriver() {

                URI uri = URI.create("neo4j://localhost:7687");
                String us = "neo4j";
                String pw = "12345678";
                Driver driver = GraphDatabase.driver(uri,
                                AuthTokens.basic(us, pw));

                return driver;
        }
}
```

Driver configuration

Certificate Type	Neo4j Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Unencrypted	neo4j	neo4j	bolt
Encrypted with Full Certificate	neo4j+s	neo4j+s	bolt+s
Encrypted with Self-Signed Certificate	neo4j+ssc	neo4j+ssc	bolt+ssc
Neo4j AuraDB	neo4j+s	N/A	N/A

Choosing your Scheme

- You will use a variation of the neo4j scheme within your connection string.
  - neo4j - Creates an unencrypted connection to the DBMS. If you are connecting to a local DBMS or have not explicitly turned on encryption then this is most likely the option you are looking for.
  - neo4j+s - Creates an encrypted connection to the DBMS. The driver will verify the authenticity of the certificate and fail to verify connectivity if there is a problem with the certificate.
  - neo4j+ssc - Creates an encrypted connection to the DBMS, but will not attempt to verify the authenticity of the certificate.

Choosing your Scheme

- Variations of the bolt scheme can be used to connect directly to a single DBMS (within a clustered environment or standalone).
- This can be useful if you have a single server configured for data science or analytics.
  - bolt - Creates an unencrypted connection directly to a single DBMS.
  - bolt+s - Creates an encrypted connection directly to a single DBMS and verify the certificate.
  - bolt+ssc - Creates an encrypted connection to directly to a single DBMS but will not attempt to verify the authenticity of the certificate.

Testing connects to the Neo4j

Using JUnit 5 to test

- Using maven

```
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.10.2</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.10.2</version>
        <scope>test</scope>
</dependency>
```

Testing connects to the Neo4j

Using JUnit 5 to test

- Using Gradle

```
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.2’
testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.10.2'
```

Testing connects to the Neo4j

- **Testing connects to the Neo4j**

```
@Test
void createDriverAndConnectToServer() {
        Driver driver = AppUtils.initDriver();
        assertNotNull(driver);
        assertNotNull(driver, "driver instantiated");
}
```

Interacting with Neo4j

You will learn how to:

- Open a Session and execute a Unit of Work within a Transaction.
- Execute Read and Write queries through the Driver.
- Consume the results returned from Neo4j.
- Handle potential errors thrown by the Driver.

Sessions and Transactions

Sessions

- Through the Driver, we open Sessions.
- A session is a container for a sequence of transactions.
- Sessions borrow connections from a pool as required and are considered lightweight and disposable.
- When the Driver connects to the database, it opens up multiple TCP connections that can be borrowed by the session.
- A query may be sent over multiple connections, and results may be received by the driver over multiple connections.

Sessions and Transactions

Sessions

- To open a new session, call the session() method on the driver.

```
Session session = driver.session();
```

- The Session needs to be closed again, fortunately it's an auto-closeable, so that a try-with-resources construct works well for us, where the session is automatically closed when the scope of the try ends.
- Use a session within a try-with-resources

```
try (var session = driver.session()) {
        // Do something with the session...
}
```

Sessions and Transactions

Sessions

- The session() method takes an optional configuration argument, which can be used to set the database to run any queries against in a multi-database setup, and the default access mode for any queries run within the transaction.
- The default access mode is set to WRITE, but this can be overwritten by explicitly calling the executeRead() or executeWrite() methods.

```
var sessionConfig = SessionConfig.builder()
        .withDefaultAccessMode(AccessMode.WRITE)
        .withDatabase("people")
        .build();
try (var session = driver.session(sessionConfig)) {
          //…
}
```

Transactions

- Through a Session, we can run one or more Transactions.
- A transaction comprises a unit of work performed against a database. It is treated in a coherent and reliable way, independent of other transactions.
- ACID TRANSACTIONS

  A transaction, by definition, must be:
  - atomic,
  - consistent,
  - isolated, and
  - durable.
- Many developers are familiar with ACID transactions from their work with relational databases, and as such the ACID consistency model has been the norm for some time.

Transactions

There are three types of transaction exposed by the driver:

- Auto-commit Transactions
- Read Transactions
- Write Transactions

Auto-commit Transactions

- Auto-commit transactions are a single unit of work that are immediately executed against the DBMS and acknowledged immediately.
- You can run an auto-commit transaction by calling the run() method on the session object, passing in a Cypher statement as a string and optionally an object containing a set of parameters.

```
var query = "MATCH () RETURN count(*) AS count";
var params = Values.parameters();
// Run a query in an auto-commit transaction
var res = session.run(query, params).single().get("count").asLong();
```

Read Transactions

- When you intend to read data from Neo4j, you should execute a Read Transaction.
- The session provides an executeRead() method, which expects a single parameter, a callback function that represents the unit of work.
- The function will accept a single parameter, a Transaction object, on which you can call the tx.run() method with two arguments: the Cypher statement as a string and an optional set of query parameters.

```
var res = session.readTransaction(tx -> {
return tx.run("\"MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
            WHERE m.title = $title // (1)
            RETURN p.name AS name
            LIMIT 10\"",
      Values.parameters("title", "Arthur") // (2)
).list(r -> r.get("name").asString());
```

Write Transactions

- If you intend to write data to the database, you should execute a Write Transaction.
- If anything goes wrong within of the unit of work or there is a problem on Neo4j's side, the transaction will be automatically rolled back and the database will remain in its previous state.

```
session.writeTransaction(tx -> {
   return tx.run(
         "CREATE (p:Person {name: $name})",
         Values.parameters("name", "Michael")).consume();
});
```

Manually Creating Transactions

```
var tx = session.beginTransaction();
```

- This returns a Transaction object identical to the one passed in to the unit of work function when calling executeRead() or executeWrite().
- You can commit a transaction by calling the tx.commit() method, or roll back the transaction by calling tx.rollback().

```
try {
        // Perform an action
        tx.run(query, params);
        // Commit the transaction
        tx.commit();
} catch (Exception e) {
        // If something went wrong, rollback the transaction
        tx.rollback();
}
```

Closing the Session

- Usually the session is auto-closed by the try-with-resources setup
- Only if you manage/pass around the session manually, you need to close it explicitly by calling the close() method to release any resources held by that session.

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

Processing Results

The Neo4j Java Driver provides you with three APIs for consuming results:

- Synchronous API
- Async API
- Reactive API

The Three APIs

- The most common and straightforward method of consuming results is with the synchronous API.
- When using session.run(), tx.run(), or one of the two transaction functions, the query will return a Result object that you can process incrementally and then return the results of that processing.
- For the asynchronous and reactive APIs you need to use different entry-points and API methods and helpers like a reactive framework. In return you get more efficient resource usage in the database, middleware and client by using the non-synchronous APIs.

Example 1. Synchronous API

```
try (var session = driver.session()) {
    var res = session.readTransaction(tx -> tx.run(
            "MATCH (p:Person) RETURN p.name AS name LIMIT 10").list());
    res.stream()
            .map(row -> row.get("name"))
            .forEach(System.out::println);
} catch (Exception e) {
    // There was a problem with the
    // database connection or the query
    e.printStackTrace();
```

Example 2. Async API

```
var session = driver.asyncSession();
session.readTransactionAsync(tx -> tx.runAsync(
            "MATCH (p:Person) RETURN p.name AS name LIMIT 10")
      .thenApplyAsync(res -> res.listAsync(row -> row.get("name")))
      .thenAcceptAsync(System.out::println)
      .exceptionallyAsync(e -> {
          e.printStackTrace();
          return null;
      })
);
```

Example 3. Reactive API

```
Flux.usingWhen(Mono.fromSupplier(driver::rxSession),
    session -> session.readTransaction(tx -> {
        var rxResult = tx.run(
                "MATCH (p:Person) RETURN p.name AS name LIMIT 10");
        return Flux
            .from(rxResult.records())
            .map(r -> r.get("name").asString())
            .doOnNext(System.out::println)
            .then(Mono.from(rxResult.consume()));
    }
    ), RxSession::close);
```

The Result

- The Result object, contains the records received by the Driver along with a set of additional meta data about the query execution and results.
- An individual row of results is referred to as a Record, and can be accessed from the result various ways, as Iterator<Record> and via the stream(), the list() or single() methods.
- A Record refers to the keyed set of values specified in the RETURN portion of the statement.
- If no RETURN values are specified, the query will not return any results, and record results will be empty or throw an error in the case of single().
- Additional meta data about the result and query is accessible from Result too

Records

- You can access the records returned by the query through several means. A Result is an Iterator<Record> there are stream() and list() accessors for streaming and materialization/conversion.
- Iterating over Records

```
Record row = res.single();
List<Record> rows = res.list();
Stream<Record> rowStream = res.stream();
while (res.hasNext()) {
    var next = res.next();
}
```

Records

- Accessing record column values
  - column names: row.keys();
  - check for existence: row.containsKey("movie");
  - number of columns: row.size();
  - get a numeric value
    - Number count = row.get("movieCount").asInt(0);
  - boolean isDirector = row.get("isDirector").asBoolean();
  - get node: row.get("movie").asNode();

Result Summary

The meta data ResultSummary accessed from Result.consume() include:

- statistics on how many nodes and relationships were created, updated, or deleted as a result of the query,
- the query type
- database and server info
- query plan with and without profile
- notifications

The Neo4j Type System

- At this point, we should take a look at the Cypher type system. Despite Neo4j being written in *Java (the j in Neo4j stands for Java after all)*, there are some discrepancies between the types available in Cypher and native Java types.
- Some values like strings, numbers, booleans, dates, and nulls map directly to Java types but more complex types like nodes, relationship, points, durations need special handling.
- Java Types to Neo4j/Cypher Types

The Neo4j Type System

Java Type	Neo4j Cypher Type	Java Type	Neo4j Cypher Type
null,	null	Time	Time
List	List, Array	LocalTime	LocalTime
Map	Map	DateTime	DateTime
Boolean	Boolean	LocalDateTime	LocalDateTime
Long	Integer	IsoDuration	Duration
Double	Float	Point	Point
String	String	Node	Node
byte[]	byte[]	Relationship	Relationship
LocalDate	LocalDate	Path	Path

Nodes & Relationships

- Nodes and Relationships are both returned as similar types with a common superclass Entity.
- As an example, let's take the following code snippet:

```
// Execute a query within a read transaction
Result res = session.readTransaction(tx -> tx.run("""
            MATCH path = (person:Person)-[actedIn:ACTED_IN]->(movie:Movie)
            RETURN path, person, actedIn, movie,
                size ( (person)-[:ACTED]->() ) as movieCount,
                exists { (person)-[:DIRECTED]->() } as isDirector
            LIMIT 1
        """));
```

Nodes

- We can retrieve the person value using the .get() method on the row and then turn it into a Node via asNode().

```
// Get a node
Node person = row.get("person").asNode();
```

- The value assigned to the person variable will be the instance of a Node. Node is a type provided by the Neo4j Java Driver to represent the information held in Neo4j for a node.
- An instance of a Node has three parts:

```
var nodeId = person.id();
var labels = person.labels();
var properties = person.asMap();
```

Relationships

- Relationship objects are also Entity instances, they also include an id, a type and properties.

```
var actedIn = row.get("actedIn").asRelationship();
var relId = actedIn.id();
String type = actedIn.type();
var relProperties = actedIn.asMap();
var startId = actedIn.startNodeId();
var endId = actedIn.endNodeId();
```

Paths

If you return a path of nodes and relationships, they will be returned as an instance of a Path.

```
Path path = row.get("path").asPath();
Node start = path.start();
Node end = path.end();
var length = path.length();
Iterable<Path.Segment> segments = path;
Iterable<Node> nodes = path.nodes();
Iterable<Relationship> rels = path.relationships();
```

Path Segments

- A path is split into segments representing each relationship in the path. For example, say we have a path of (p:Person)-[:ACTED_IN]->(m:Movie)-[:IN_GENRE]->(g:Genre), there would be two segments.

    1. (p:Person)-[:ACTED_IN]->(m:Movie)
    2. (m:Movie)-[:IN_GENRE]->(g:Genre)

- The PathSegment object has three properties:
    - relationship - A Relationship object representing that part of the path.
    - start - start node for this path segment *
    - end - end node for this path segment *

Converting these values

- There may be times when you need to convert many Neo4j types back into native Java types. For example, when retrieving a set of properties.
- For this the Value and Record type has three functions
  - asObject() recursively converts a Value into the appropriate Java objects
  - asMap() converts a Value into a Java Map, it can take a callback function to customize conversion of individual keys and values
  - list() converts a Value into a Java List, it can take a callback function to customize conversion of individual values
- The function are recursive, and will handle nested objects and arrays.

Additional helper functions

Value has additional helper functions

- isTrue and isFalse for boolean values
- isNull for null checks
- isEmpty for lists and maps