The Neo4j Java Reference v5.4

Table of Contents

The Java Reference contains information on advanced Java-centric usage of Neo4j.

It covers the following topics:

- Extending Neo4j — How to build unmanaged extensions and procedures.
- Using Neo4j embedded in Java applications — Instructions on embedding Neo4j in .
- Traversal Framework — A walkthrough of the traversal framework.
- Transaction management — Details on transaction semantics in Neo4j.
- JMX metrics — How to monitor Neo4j with JMX and a reference of available metrics.

	You might want to keep the Neo4j Javadocs (Neo4j Java API Documentation) handy while reading.

Who should read this?

The Java Reference is written for advanced Java developers who want to extend Neo4j's functionality or embed Neo4j in their software.

Extending Neo4j

This section describes how to extend Neo4j and Cypher using procedures, functions, and plugins. It introduces different methods to extend the standard Neo4j functionality and explains how to set up remote debugging.

Neo4j provides the following methods to extend the standard functionality:

- **Procedures and functions** extend the capabilities of the Cypher query language.
- **Authentication and authorization plugins** extend the Neo4j security framework.
- **Server extensions** enable new surfaces to be created in the HTTP API.

Writing extensions requires the user to be familiar with Java or other JVM programming languages, and to have an environment set up for compiling such code.

Best practices

It is important to consider any security implications of deploying customized code. Refer to the Operations Manual → Securing Extensions for details on best practices for securing user-defined procedures and functions.

Since you will be running custom-built code and Neo4j in the same JVM, there are a few things you should keep in mind:

- Do not create or retain more objects than you strictly need to. Large caches in particular tend to promote more objects to the old generation, thus increasing the need for expensive full garbage collections.
- Do not use internal Neo4j APIs. They are internal to Neo4j and subject to change without notice, which may break or change the behavior of your code.
- If possible, avoid using Java object serialization or reflection in your code or in any runtime dependency that you use. If you cannot avoid using Java object serialization and reflection, ensure that the `-XX:+TrustFinalNonStaticFields` JVM flag is disabled in `neo4j.conf`.

Neo4j customized code

User-defined procedures and *user-defined functions* are mechanisms that enable you to extend Neo4j by writing customized code, which can be invoked directly from Cypher. This is the preferred means for extending Neo4j.

Examples of use cases for procedures and functions are:

- To provide access to functionality that is not available in Cypher.
- To provide access to third-party systems.
- To perform graph-global operations, such as counting connected components or finding dense nodes.
- To express a procedural operation that is difficult to express declaratively with Cypher.

Procedures and functions are written in Java and compiled into JAR files. They are deployed to the database by dropping that JAR file into the *plugins* directory on each standalone or clustered server. For the location of the *plugins* directory, refer to Operations Manual → Default file locations. The database must be restarted on each server to pick up new procedures and functions.

Procedures and functions can take arguments and return results. In addition, procedures can perform write operations on the database.

Type	Description	Syntax	Read/Write	Cardinality
`Procedure`	For each row, the procedure takes parameters and returns multiple results.	`CALL abc(...)`	Update allowed.	Changes cardinality similarly to a `MATCH` clause (0, 1, or many).
`Scalar function`	For each row, the function takes parameters and returns a single result.	`abc(...)`	Read-only.	Maintains cardinality, one for one.
`Aggregating function`	Consumes many rows and produces an aggregated result.	`WITH abc(...)`	Read-only.	Reduces cardinality, many down to one.

Neo4j also comes bundled with a number of *built-in* procedures and functions.

The available built-in procedures vary depending on edition and mode, as described in Operations Manual → Procedures. Running `SHOW PROCEDURES` displays the full list of procedures available in your Neo4j DBMS, including user-defined procedures.

The built-in functions are described in Cypher Manual → Functions. Running `SHOW FUNCTIONS` displays the full list of all the functions available in your Neo4j DBMS, including user-defined functions.

Setting up a plugin project

You can set up a project for extending Neo4j with a user-defined procedure, build the project, and deploy the procedure to a Neo4j instance. The same steps can be used for user-defined functions.

> The example described in this section is available on GitHub (neo4j-examples/neo4j-procedure-template).

Set up a project with Maven

A project can be set up in any way that allows for compiling a procedure and producing a JAR file.

Below are the main parts of the example configuration, using the Maven build system.

Excerpts from the Maven pom.xml file

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.neo4j.example</groupId>
  <artifactId>procedure-template</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <packaging>jar</packaging>
  <name>Neo4j Procedure Template</name>
  <description>A template project for building a Neo4j Procedure</description>

  <properties>
    <neo4j.version>5.4.0</neo4j.version>
  </properties>
```

Build dependencies

Add a dependency section that includes the procedure and function APIs, which procedures and functions use at runtime.

The scope is set to `provided` because once the procedure is deployed to a Neo4j instance, this dependency is provided by Neo4j. If non-Neo4j dependencies are added to the project, their scope should normally be `compile`.

```
<dependency>
 <groupId>org.neo4j</groupId>
 <artifactId>neo4j</artifactId>
 <version>${neo4j.version}</version>
 <scope>provided</scope>
</dependency>
```

Add dependencies that are necessary for testing the procedure.

Neo4j Harness, a utility that allows for starting a lightweight Neo4j instance. It is used to start Neo4j with a specific procedure or function deployed, which greatly simplifies testing.

Neo4j Java Driver, used to send Cypher statements that call the procedure or function.

JUnit, a common Java test framework.

```xml
<dependency>
 <groupId>org.neo4j.test</groupId>
 <artifactId>neo4j-harness</artifactId>
 <version>${neo4j.version}</version>
 <scope>test</scope>
</dependency>

<dependency>
 <groupId>org.neo4j.driver</groupId>
 <artifactId>neo4j-java-driver</artifactId>
 <version>5.3</version>
 <scope>test</scope>
</dependency>

<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.12</version>
 <scope>test</scope>
</dependency>
```

Build steps

The steps that Maven will go through to build the project.

The goal is first to *compile* the source, then to *package* it in a JAR that can be deployed to a Neo4j instance.

The Maven Shade plugin is used to package the compiled procedure. It also includes all dependencies in the package, unless the dependency scope is set to test or *provided*.

Once the procedure has been deployed to the *plugins* directory of each Neo4j instance and the instances have restarted, the procedure is available for use.

```xml
<build>
<plugins>
 <plugin>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
     <source>11</source>
     <target>11</target>
   </configuration>
 </plugin>
 <plugin>
   <artifactId>maven-shade-plugin</artifactId>
   <executions>
     <execution>
       <phase>package</phase>
       <goals>
         <goal>shade</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
</plugins>
</build>
```

Values and types

The *input* and *output* to and from a procedure or a function must be one of the supported types, as described in Cypher Manual → Values and types.

Composite types are supported via:

- `List<T>`, where `T` is one of the supported types, and
- `Map<String, Object>`, where the values in the map must have one of the supported types.

The use of `Object` is supported for the case where the type is not known beforehand. Note, however, that the actual value must still have one of the aforementioned types.

Table 1. Supported Cypher types and their equivalents in Java

Cypher type	Java type
`String`	`String`
`Integer`	`Long`
`Float`	`Double`
`Boolean`	`Boolean`
`Point`	`org.neo4j.graphdb.spatial.Point`
`Date`	`java.time.LocalDate`
`Time`	`java.time.OffsetTime`
`LocalTime`	`java.time.LocalTime`
`DateTime`	`java.time.ZonedDateTime`
`LocalDateTime`	`java.time.LocalDateTime`
`Duration`	`java.time.temporal.TemporalAmount`
`Node`	`org.neo4j.graphdb.Node`
`Relationship`	`org.neo4j.graphdb.Relationship`
`Path`	`org.neo4j.graphdb.Path`

For more details, see the Neo4j Javadocs `org.neo4j.procedure.Procedure`.

There are two cases where more than one Java type is mapped to a single Cypher type. When this happens, type information is lost. If the following objects are returned from procedures, the original types cannot be recreated:

- A Cypher `Duration` is created when either `java.time.Duration` or `java.time.Period` is provided. If `Duration` is returned, only the common interface `java.time.temporal.TemporalAmount` remains.
- A Cypher `DateTime` is created when `java.time.OffsetDateTime` is provided. If `DateTime` is returned, it is converted into `java.time.ZonedDateTime`.

User-defined procedures

A *user-defined procedure* is a mechanism that enables you to extend Neo4j by writing customized code, which can be invoked directly from Cypher. Procedures can take arguments, perform operations on the database, and return results. For a comparison between user-defined procedures, functions, and aggregation functions see Neo4j customized code.

User-defined procedures requiring execution on the system database need to include the annotation `@SystemProcedure` or they will be classed as user database procedure.

Call a procedure

To call a user-defined procedure, use a Cypher `CALL` clause. The procedure name must be fully qualified, so a procedure named `findDenseNodes` defined in the package `org.neo4j.examples` could be called using:

```
CALL org.neo4j.examples.findDenseNodes(1000)
```

`CALL` may be the only clause within a Cypher statement or may be combined with other clauses. Arguments can be supplied directly within the query or taken from the associated parameter set. For full details, see the documentation in Cypher Manual → `CALL` procedure.

Create a procedure

Make sure you have read and followed the preparatory setup instructions in Setting up a plugin project.

The example discussed below is available as a repository on GitHub. To get started quickly you can fork the repository and work with the code as you follow along in the guide below.

First, decide what the procedure should do, then write a test that proves that it does it right. Finally, write a procedure that passes the test.

Integration tests

The test dependencies include *Neo4j Harness* and *JUnit*. These can be used to write integration tests for procedures. The tests should start a Neo4j instance, load the procedure, and execute queries against it.

A template for testing a procedure that accesses Neo4j's full-text indexes from Cypher.

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.graphdb.factory.GraphDatabaseSettings;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;
import static org.neo4j.driver.v1.Values.parameters;

public class ManualFullTextIndexTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

            // This is the Procedure to test
            .withProcedure( FullTextIndex.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // In a try-block, make sure you close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withoutEncryption
().toConfig() ) )
        {

            // Given you've started Neo4j with the FullTextIndex procedure class
            //       which your 'neo4j' rule does.
            Session session = driver.session();

            // And given you have a node in the database
            long nodeId = session.run( "CREATE (p:User {name:'Brookreson'}) RETURN id(p)" )
                    .single()
                    .get( 0 ).asLong();

            // When you use the index procedure to index a node
            session.run( "CALL example.index($id, ['name'])", parameters( "id", nodeId ) );

            // Then you can search for that node with Lucene query syntax
            StatementResult result = session.run( "CALL example.search('User', 'name:Brook*')" );
            assertThat( result.single().get( "nodeId" ).asLong(), equalTo( nodeId ) );
        }
    }
}
```

Define a procedure

With the test in place, write a procedure that fulfills the expectations of the test. The full example is available in the Neo4j Procedure Template repository.

Particular things to note:

- All procedures are annotated `@Procedure`.
- The procedure annotation can take three optional arguments: `name`, `mode`, and `eager`.
  - `name` is used to specify a different name for the procedure than the default generated, which is `class.path.nameOfMethod`. If `mode` is specified, `name` must be specified as well.
  - `mode` is used to declare the types of interactions that the procedure performs. A procedure fails if it attempts to execute database operations that violate its mode. The default `mode` is `READ`. The following modes are available:

        ▪ `READ` — This procedure only performs read operations against the graph.
        ▪ `WRITE` — This procedure performs read and write operations against the graph.
        ▪ `SCHEMA` — This procedure performs operations against the schema, i.e. create and drop indexes and constraints. A procedure with this mode can read graph data, but not write.
        ▪ `DBMS` — This procedure performs system operations such as user management and query management. A procedure with this mode is not able to read or write graph data.
    ◦ `eager` is a boolean setting defaulting to `false`. If it is set to `true`, the Cypher planner plans an extra `eager` operation before calling the procedure. This is useful in cases where the procedure makes changes to the database in a way that could interact with the operations preceding the procedure. For example:

```
MATCH (n)
WHERE n.key = 'value'
WITH n
CALL deleteNeighbours(n, 'FOLLOWS')
```

      This query can delete some of the nodes that are matched by the Cypher query, and the `n.key` lookup will fail. Marking this procedure as `eager` prevents this from causing an error in Cypher code. However, it is still possible for the procedure to interfere with itself by trying to read entities it has previously deleted. It is the responsibility of the procedure author to handle that case.

- The context of the procedure, which is the same as each resource that the procedure wants to use, is annotated `@Context`.

> The correct way to signal an error from within a procedure is to throw `RuntimeException`.

Injectable resources

When writing procedures, some resources can be injected into the procedure from the database. To inject these, use the `@Context` annotation. The classes that can be injected are:

- `Log`
- `TerminationGuard`
- `GraphDatabaseService`
- `Transaction`

All of the above classes are considered safe and future-proof and do not compromise the security of the database. Several unsupported (restricted) classes can also be injected and can be changed with little or no notice. Procedures written to use these restricted APIs are not loaded by default, and you need to use the `dbms.security.procedures.unrestricted` to load unsafe procedures. Read more about this config setting in Operations Manual → Securing extensions.

User-defined functions

User-defined functions are simpler forms of procedures that return a single value and are read-only. Although they are less powerful in capability, they are often easier to use and more efficient than procedures for many common tasks. For a comparison between user-defined procedures, functions, and aggregation functions see Neo4j customized code.

Call a user-defined function

User-defined functions are called in the same way as any other Cypher function. The function name must be fully qualified, so a function named `join` defined in the package `org.neo4j.examples` could be called using:

```
MATCH (p: Person) WHERE p.age = 36
RETURN org.neo4j.examples.join(collect(p.names))
```

Create a function

User-defined functions are created similarly to how procedures are created. But unlike procedures, they are annotated with `@UserFunction` and return a single value instead of a stream of values.

See Values and types for details on values and types.

For more details, see the Neo4j Javadocs for `org.neo4j.procedure.UserFunction`.

> The correct way to signal an error from within a function is to throw `RuntimeException`.

```
package example;

import org.neo4j.procedure.Name;
import org.neo4j.procedure.Procedure;
import org.neo4j.procedure.UserFunction;

public class Join
{
    @UserFunction
    @Description("example.join(['s1','s2',...], delimiter) - join the given strings with the given
delimiter.")
    public String join(
            @Name("strings") List<String> strings,
            @Name(value = "delimiter", defaultValue = ",") String delimiter) {
        if (strings == null || delimiter == null) {
            return null;
        }
        return String.join(delimiter, strings);
    }
}
```

Integration tests

Tests for user-defined functions are created in the same way as those for procedures.

A template for testing a user-defined function that joins a list of strings.

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;

public class JoinTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

            // This is the function to test
            .withFunction( Join.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // This is in a try-block, to make sure you close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withEncryptionLevel(
Config.EncryptionLevel.NONE ).toConfig() ) )
        {
            // Given
            Session session = driver.session();

            // When
            String result = session.run( "RETURN example.join(['Hello', 'World']) AS result").single().
get("result").asString();

            // Then
            assertThat( result, equalTo( "Hello,World" ) );
        }
    }
}
```

User-defined aggregation functions

User-defined aggregation functions are functions that aggregate data and return a single result. For a comparison between user-defined procedures, functions, and aggregation functions, see Neo4j customized code.

Call an aggregation function

User-defined aggregation functions are called in the same way as any other Cypher aggregation function. The function name must be fully qualified, so a function named `longestString` defined in the package `org.neo4j.examples` could be called using:

```
MATCH (p: Person) WHERE p.age = 36
RETURN org.neo4j.examples.longestString(p.name)
```

Writing a user-defined aggregation function

User-defined aggregation functions are annotated with `@UserAggregationFunction`. The annotated function must return an instance of an aggregator class. An aggregator class contains one method annotated with `@UserAggregationUpdate` and one method annotated with `@UserAggregationResult`. The method annotated with `@UserAggregationUpdate` will be called multiple times and enables the class to aggregate data. When the aggregation is done, the method annotated with `@UserAggregationResult` will be called once and the result of the aggregation will be returned.

See Values and types for details on values and types.

For more details, see the Neo4j Javadocs for `org.neo4j.procedure.UserAggregationFunction`.

> The correct way to signal an error from within an aggregation function is to throw `RuntimeException`.

```
package example;

import org.neo4j.procedure.Description;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.UserAggregationFunction;
import org.neo4j.procedure.UserAggregationResult;
import org.neo4j.procedure.UserAggregationUpdate;

public class LongestString
{
    @UserAggregationFunction
    @Description( "org.neo4j.function.example.longestString(string) - aggregates the longest string found"
)
    public LongStringAggregator longestString()
    {
        return new LongStringAggregator();
    }

    public static class LongStringAggregator
    {
        private int longest;
        private String longestString;

        @UserAggregationUpdate
        public void findLongest(
                @Name( "string" ) String string )
        {
            if ( string != null && string.length() > longest)
            {
                longest = string.length();
                longestString = string;
            }
        }

        @UserAggregationResult
        public String result()
        {
            return longestString;
        }
    }
}
```

Integration tests

Tests for user-defined aggregation functions are created in the same way as those for normal user-defined functions.

A template for testing a user-defined aggregation function that finds the longest string.

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;

public class LongestStringTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

            // This is the function to test
            .withAggregationFunction( LongestString.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // This is in a try-block, to make sure you close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withEncryptionLevel(
Config.EncryptionLevel.NONE ).toConfig() ) )
        {
            // Given
            Session session = driver.session();

            // When
            String result = session.run( "UNWIND ["abc", "abcd", "ab"] AS string RETURN
example.longestString(string) AS result").single().get("result").asString();

            // Then
            assertThat( result, equalTo( "abcd" ) );
        }
    }
}
```

Authentication and authorization plugins

Neo4j provides authentication and authorization plugin interfaces to support real-world deployment scenarios not covered by native users or the built-in configuration-based LDAP connector.

The SPI (Service Provider Interface) lives in the `com.neo4j.server.security.enterprise.auth.plugin.spi` package.

Customized-built plugins have access to the <neo4j-home> directory in case you want to load any customized settings from a file located there. Plugins can also write to the security event log.

Authentication plugin

The authentication plugin implements the `AuthenticationPlugin` interface with the `authenticate` method.

The example below shows a minimal authentication plugin that checks for a Neo4j user with a Neo4j password:

```
@Override
public AuthenticationInfo authenticate( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();

    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials, "neo4j".toCharArray() ) )
    {
        return (AuthenticationInfo) () -> "neo4j";
    }
    return null;
}
```

Authorization plugin

The authorization plugin implements the `AuthorizationPlugin` interface with the `authorize` method.

The example below shows a minimal authorization plugin that assigns the reader role to a user named `neo4j`:

```
@Override
public AuthorizationInfo authorize( Collection<PrincipalAndProvider> principals )
{
    if ( principals.stream().anyMatch( p -> "neo4j".equals( p.principal() ) ) )
    {
        return (AuthorizationInfo) () -> Collections.singleton( PredefinedRoles.READER );
    }
    return null;
}
```

Note the usage of the helper class `PredefinedRole`.

Simplified combined plugin

There is also a simplified combined plugin interface `AuthPlugin`, which provides both authentication and authorization in a single method called `authenticateAndAuthorize`.

The example below shows a combined plugin verifying `neo4j`/`neo4j` credentials and returning reader role authorization:

```
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();

    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials, "neo4j".toCharArray() ) )
    {
        return AuthInfo.of( "neo4j", Collections.singleton( PredefinedRoles.READER ) );
    }
    return null;
}
```

Extendable platform

Neo4j provides an extendable platform as some user deployment scenarios may not be easily configured through the standard LDAP connector. One known complexity is integrating with the LDAP user directory where groups have users as a member and not the other way around.

The example below first searches for a group that the user is a member of, and then maps that group to the Neo4j role by calling the customized-built `getNeo4jRoleForGroupId` method:

```java
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken ) throws AuthenticationException
{
    try
    {
        String username = authToken.principal();
        char[] password = authToken.credentials();

        LdapContext ctx = authenticate( username, password );
        Set<String> roles = authorize( ctx, username );

        return AuthInfo.of( username, roles );
    }
    catch ( NamingException e )
    {
        throw new AuthenticationException( e.getMessage() );
    }
}

private LdapContext authenticate( String username, char[] password ) throws NamingException
{
    Hashtable<String,Object> env = new Hashtable<>();
    env.put( Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory" );
    env.put( Context.PROVIDER_URL, "ldap://0.0.0.0:10389" );

    env.put( Context.SECURITY_PRINCIPAL, String.format( "cn=%s,ou=users,dc=example,dc=com", username ) );
    env.put( Context.SECURITY_CREDENTIALS, password );

    return new InitialLdapContext( env, null );
}

private Set<String> authorize( LdapContext ctx, String username ) throws NamingException
{
    Set<String> roleNames = new LinkedHashSet<>();

    // Set up our search controls
    SearchControls searchCtls = new SearchControls();
    searchCtls.setSearchScope( SearchControls.SUBTREE_SCOPE );
    searchCtls.setReturningAttributes( new String[]{GROUP_ID} );

    // Use a search argument to prevent potential code injection
    Object[] searchArguments = new Object[]{username};

    // Search for groups that has the user as a member
    NamingEnumeration result = ctx.search( GROUP_SEARCH_BASE, GROUP_SEARCH_FILTER, searchArguments,
searchCtls );

    if ( result.hasMoreElements() )
    {
        SearchResult searchResult = (SearchResult) result.next();

        Attributes attributes = searchResult.getAttributes();
        if ( attributes != null )
        {
            NamingEnumeration attributeEnumeration = attributes.getAll();
            while ( attributeEnumeration.hasMore() )
            {
                Attribute attribute = (Attribute) attributeEnumeration.next();
                String attributeId = attribute.getID();
                if ( attributeId.equalsIgnoreCase( GROUP_ID ) )
                {
                    // Found a group that the user is a member of. See if it has a role mapped to it
                    String groupId = (String) attribute.get();
                    String neo4jGroup = getNeo4jRoleForGroupId( groupId );
                    if ( neo4jGroup != null )
                    {
                        // Yay! Add it to your set of roles
                        roleNames.add( neo4jGroup );
                    }
                }
            }
        }
    }
    return roleNames;
}
```

💡	For more information and other plugin examples, see https://github.com/neo4j/neo4j-example-auth-plugins.

Full-text index analyzer providers

Full-text indexes always have an analyzer, which describes how text is analyzed for indexing and querying. The analyzer breaks the text up into smaller tokens and processes these tokens using filters. The filters can do different things, such as removing stop-words (e.g., *the* and *is*), stemming the tokens, or turning them into lower-case.

Which analyzer to use, depends on what you want to use the index for. For example, if the text being indexed belongs to a special domain, such as email addresses, you would want an analyzer specific to that domain. Or if the text is always in a particular language, such as Russian, you could use an analyzer specific to that language.

Neo4j comes with a number of built-in analyzers. The full list of analyzers is available by calling the `db.index.fulltext.listAvailableAnalyzers()` procedure. The procedure returns the analyzer name, a short description, and the full list of stop-words used by the analyzer.

The default analyzer is the `standard-no-stop-words` analyzer. This default can be changed with the `dbms.index.fulltext.default_analyzer` setting. This setting only takes effect when a full-text index is created. Once a full-text index has been created, it remembers the analyzer in its index-specific settings.

It is possible to extend the available analyzers in Neo4j by implementing `AnalyzerProvider`. The `AnalyzerProvider` acts as a factory, which builds the concrete Lucene `Analyzer` instance used by the index. The following example creates a customized analyzer:

```
public class CustomAnalyzerProvider extends AnalyzerProvider ①
{
    public CustomAnalyzerProvider()                          ②
    {
        super( "custom-analyzer" );                          ③
    }

    @Override
    public Analyzer createAnalyzer()                         ④
    {
        try
        {
            return CustomAnalyzer.builder()                  ⑤
                    .withTokenizer( StandardTokenizerFactory.class )
                    .addTokenFilter( LowerCaseFilterFactory.class )
                    .addTokenFilter( StopFilterFactory.class, "ignoreCase", "false", "words",
"stopwords.txt", "format", "wordset" )
                    .build();
        }
        catch ( IOException e )
        {
            throw new UncheckedIOException( e );
        }
    }
}
```

① The `CustomAnalyzerProvider` class must be `public`, and it must extend the `org.neo4j.graphdb.schema.AnalyzerProvider` class.

② The `CustomAnalyzerProvider` class must additionally have a `public` constructor that takes no arguments. Without this constructor, the new analyzer provider will not be loaded by Neo4j. There will be no warnings logged when an analyzer provider is ignored because of this reason, so take care.

③ The constructor must then call its super-constructor, and give the name of the `custom-analyzer` provider as an argument. This is the name that will be used to refer to this analyzer provider when configuring indexes.

④ Lastly, the `createAnalyzer` method must be implemented. This method creates and returns the concrete `Analyzer` instance, that the indexes will use. If this method returns `null` or throws an exception, the index will be marked as FAILED.

⑤ This example creates instances of Lucene's `CustomAnalyzer`. You can, however, create and return anything that extends the `org.apache.lucene.analysis.Analyzer` class.

Follow the guide in Setting up a plugin project, to learn how to package the customized `AnalyzerProvider` in a JAR file that can be integrated into Neo4j.

The analyzer providers are picked up by Neo4j via service loading. This means that in addition to having implemented the class, you must also add the fully qualified class name to a service file, and put that file on the classpath. These service files are usually included in the JAR file that contains the Neo4j extensions. In a typical Maven project, such as the one created by Setting up a plugin project, the directory structure would look like this:

```
project/
   src/
      main/
         java/
            my_package/
               CustomAnalyzerProvider.java ①
         resources/
            META-INF/
               services/
                  org.neo4j.graphdb.schema.AnalyzerProvider ②
```

① This is the `CustomAnalyzerProvider` from our previous code example.

② This is the service loader file. It is a plain-text file that contains a line, with the fully qualified name, for each `AnalyzerProvider` implemented in our project. In this case, it contains a single line: `my_package.CustomAnalyzerProvider`.

For the `META-INF/services` resources to be handled correctly by the `maven-shade-plugin`, it may be necessary to include a service resource transformation to the plug-in configuration. This is an example of what that may look like:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Consult the documentation on the Maven Shade Plugin for more details on this step.

Unmanaged server extensions

Introduction

Unmanaged server extensions are used if you want to have a finer-grained level of control over your application's interactions with Neo4j than Cypher provides.

> This is a sharp tool, that enables users to deploy arbitrary JAX-RS classes to the server so be careful when using it. In particular, it is possible to consume lots of heap space on the server and degrade performance. If in doubt, please ask for help via one of the community channels.

The first step when writing an unmanaged extension is to create a project which includes dependencies to the Neo4j core JARs. In Maven, this would be achieved by adding the following lines to the POM file:

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>5.4.0</version>
    <scope>provided</scope>
</dependency>
```

Now you are ready to write your extension.

In your code, you interact with Neo4j using `DatabaseManagementService`, which you can access by using the `@Context` annotation. The following example serves as a template on which you can base your extension:

```
@Path( "/helloworld" )
public class HelloWorldResource
{
    private final DatabaseManagementService dbms;

    public HelloWorldResource( @Context DatabaseManagementService dbms )
    {
        this.dbms = dbms;
    }

    @GET
    @Produces( MediaType.TEXT_PLAIN )
    @Path( "/{nodeId}" )
    public Response hello( @PathParam( "nodeId" ) long nodeId )
    {
        // Do stuff with the database
        return Response.status( Status.OK ).entity( UTF8.encode( "Hello World, nodeId=" + nodeId ) ).
build();
    }
}
```

The full source code isfound at: HelloWorldResource.java

Having built your code, the resulting JAR file (and any customized dependencies) should be placed in the *$NEO4J_SERVER_HOME/plugins* directory. You also need to tell Neo4j where to look for the extension by adding some configuration in *neo4j.conf*:

```
#Comma-separated list of JAXRS packages containing JAXRS Resource, one package name for each mountpoint.
server.unmanaged_extension_classes=org.neo4j.examples.server.unmanaged=/examples/unmanaged
```

Your `hello` method responds to `GET` requests at the URI:

```
http://{neo4j_server}:{neo4j_port}/examples/unmanaged/helloworld/{node_id}
```

For example:

```
curl http://localhost:7474/examples/unmanaged/helloworld/123
```

which results in:

```
Hello World, nodeId=123
```

Streaming JSON responses

When writing unmanaged extensions, you have greater control over the amount of memory that your Neo4j queries use. If you keep too much state around, it can lead to more frequent full Garbage Collection and subsequent unresponsiveness by the Neo4j server.

A common way that state can increase, is the creation of JSON objects to represent the result of a query, which is then sent back to your application. Neo4j's Transactional Cypher HTTP endpoint (see HTTP API Docs → transactional Cypher endpoint) streams responses back to the client. For example, the following unmanaged extension streams an array of a person's colleagues:

```java
@Path("/colleagues")
public class ColleaguesResource
{
    private DatabaseManagementService dbms;
    private final ObjectMapper objectMapper;

    private static final RelationshipType ACTED_IN = RelationshipType.withName( "ACTED_IN" );
    private static final Label PERSON = Label.label( "Person" );

    public ColleaguesResource( @Context DatabaseManagementService dbms )
    {
        this.dbms = dbms;
        this.objectMapper = new ObjectMapper();
    }

    @GET
    @Path("/{personName}")
    public Response findColleagues( @PathParam("personName") final String personName )
    {
        StreamingOutput stream = new StreamingOutput()
        {
            @Override
            public void write( OutputStream os ) throws IOException, WebApplicationException
            {
                JsonGenerator jg = objectMapper.getJsonFactory().createJsonGenerator( os, JsonEncoding
.UTF8 );
                jg.writeStartObject();
                jg.writeFieldName( "colleagues" );
                jg.writeStartArray();

                final GraphDatabaseService graphDb = dbms.database( "neo4j" );
                try ( Transaction tx = graphDb.beginTx();
                      ResourceIterator<Node> persons = tx.findNodes( PERSON, "name", personName ) )
                {
                    while ( persons.hasNext() )
                    {
                        Node person = persons.next();
                        for ( Relationship actedIn : person.getRelationships( OUTGOING, ACTED_IN ) )
                        {
                            Node endNode = actedIn.getEndNode();
                            for ( Relationship colleagueActedIn : endNode.getRelationships( INCOMING,
ACTED_IN ) )
                            {
                                Node colleague = colleagueActedIn.getStartNode();
                                if ( !colleague.equals( person ) )
                                {
                                    jg.writeString( colleague.getProperty( "name" ).toString() );
                                }
                            }
                        }
                    }
                    tx.commit();
                }

                jg.writeEndArray();
                jg.writeEndObject();
                jg.flush();
                jg.close();
            }
        };

        return Response.ok().entity( stream ).type( MediaType.APPLICATION_JSON ).build();
    }
}
```

The full source code isfound at: ColleaguesResource.java

As well as depending on JAX-RS API, this example also uses Jackson — a Java JSON library. You need to add the following dependency to your Maven POM file (or equivalent):

```xml
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.2</version>
</dependency>
```

From Neo4j 3.5.15, a breaking change was introduced following an update to the Jackson dependency.

Jackson v1 is out of support and has accumulated security issues such as:

- `CVE-2017-7525`
- `CVE-2017-17485`
- `CVE-2017-15095`
- `CVE-2018-11307`
- `CVE-2018-7489`
- `CVE-2018-5968`

For further information about Jackson v2, please see the Jackson Project on GitHub.

Your `findColleagues` method now responds to `GET` requests at the URI:

```
http://{neo4j_server}:{neo4j_port}/examples/unmanaged/colleagues/{personName}
```

For example:

```
curl http://localhost:7474/examples/unmanaged/colleagues/Keanu%20Reeves
```

which results in:

```
{"colleagues":["Hugo Weaving","Carrie-Anne Moss","Laurence Fishburne"]}
```

Executing Cypher

You can execute Cypher queries by using the `GraphDatabaseService`, which is injected into the extension. For example, the following unmanaged extension retrieves a person’s colleagues using Cypher:

```
@Path("/colleagues-cypher-execution")
public class ColleaguesCypherExecutionResource
{
    private final ObjectMapper objectMapper;
    private DatabaseManagementService dbms;

    public ColleaguesCypherExecutionResource( @Context DatabaseManagementService dbms )
    {
        this.dbms = dbms;
        this.objectMapper = new ObjectMapper();
    }

    @GET
    @Path("/{personName}")
    public Response findColleagues( @PathParam("personName") final String personName )
    {
        final Map<String, Object> params = MapUtil.map( "personName", personName );

        StreamingOutput stream = new StreamingOutput()
        {
            @Override
            public void write( OutputStream os ) throws IOException, WebApplicationException
            {
                JsonGenerator jg = objectMapper.getJsonFactory().createJsonGenerator( os, JsonEncoding
.UTF8 );
                jg.writeStartObject();
                jg.writeFieldName( "colleagues" );
                jg.writeStartArray();

                final GraphDatabaseService graphDb = dbms.database( "neo4j" );
                try ( Transaction tx = graphDb.beginTx();
                      Result result = tx.execute( colleaguesQuery(), params ) )
                {
                    while ( result.hasNext() )
                    {
                        Map<String,Object> row = result.next();
                        jg.writeString( ((Node) row.get( "colleague" )).getProperty( "name" ).toString()
);
                    }
                    tx.commit();
                }

                jg.writeEndArray();
                jg.writeEndObject();
                jg.flush();
                jg.close();
            }
        };

        return Response.ok().entity( stream ).type( MediaType.APPLICATION_JSON ).build();
    }

    private String colleaguesQuery()
    {
        return "MATCH (p:Person {name: $personName })-[:ACTED_IN]->()<-[:ACTED_IN]-(colleague) RETURN
colleague";
    }
}
```

The full source code isfound at: ColleaguesCypherExecutionResource.java

Your `findColleagues` method now responds to `GET` requests at the URI:

```
http://{neo4j_server}:{neo4j_port}/examples/unmanaged/colleagues-cypher-execution/{personName}
```

For example:

```
curl http://localhost:7474/examples/unmanaged/colleagues-cypher-execution/Keanu%20Reeves
```

which results in:

```
{"colleagues": ["Hugo Weaving", "Carrie-Anne Moss", "Laurence Fishburne"]}
```

Testing your extension

Neo4j provides tools to help you write integration tests for your extensions. You can access this toolkit by adding the following test dependency to your project:

```
<dependency>
   <groupId>org.neo4j.test</groupId>
   <artifactId>neo4j-harness</artifactId>
   <version>5.4.0</version>
   <scope>test</scope>
</dependency>
```

The test toolkit provides a mechanism to start a Neo4j instance with a customized configuration and with extensions of your choice. It also provides mechanisms to specify data fixtures to include when starting Neo4j, as you can see in the following example:

```
@Path("")
public static class MyUnmanagedExtension
{
    @GET
    public Response myEndpoint()
    {
        return Response.ok().build();
    }
}

@Test
public void testMyExtension() throws Exception
{
    // Given
    HTTP.Response response = HTTP.GET( HTTP.GET( neo4j.httpURI().resolve( "myExtension" ).toString()
).location() );

    // Then
    assertEquals( 200, response.status() );
}

@Test
public void testMyExtensionWithFunctionFixture()
{
    final GraphDatabaseService graphDatabaseService = neo4j.defaultDatabaseService();
    try ( Transaction transaction = graphDatabaseService.beginTx() )
    {
        // Given
        Result result = transaction.execute( "MATCH (n:User) return n" );

        // Then
        assertEquals( 1, count( result ) );
        transaction.commit();
    }
}
```

The full source code of the example isfound at: ExtensionTestingDocIT.java

Note the use of `server.httpURI().resolve( "myExtension" )` to ensure that the correct base URI is used.

If you are using the JUnit test framework, there is a JUnit rule available as well:

```java
@Rule
public Neo4jRule neo4j = new Neo4jRule()
        .withFixture( "CREATE (admin:Admin)" )
        .withFixture( graphDatabaseService ->
        {
            try (Transaction tx = graphDatabaseService.beginTx())
            {
                tx.createNode( Label.label( "Admin" ) );
                tx.commit();
            }
            return null;
        } );

@Test
public void shouldWorkWithServer()
{
    // Given
    URI serverURI = neo4j.httpURI();

    // When you access the server
    HTTP.Response response = HTTP.GET( serverURI.toString() );

    // Then it should reply
    assertEquals(200, response.status());

    // and you have access to underlying GraphDatabaseService
    try (Transaction tx = neo4j.defaultDatabaseService().beginTx()) {
        assertEquals( 2, count(tx.findNodes( Label.label( "Admin" ) ) ));
        tx.commit();
    }
}
```

The full source code of the example isfound at: JUnitDocIT.java

Setup for remote debugging

To configure Neo4j for remote debugging sessions, the Java debugging parameters need to be passed to the Java process through the configuration. They live in the *conf/neo4j.conf* file.

To specify the parameters, you must add a line for the additional Java arguments:

```
dbms.jvm.additional=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```

This configuration starts Neo4j, ready for remote debugging attachment, at `localhost`` and port `5005`. Use these parameters to attach to the process from Eclipse, IntelliJ, or your remote debugger of choice after starting the server.

Using Neo4j embedded in Java applications

This section describes how to use Neo4j embedded in Java applications.

The following topics are:

- Embedding Neo4j in your Java application
- Hello world
- Property values
- Using indexes
- Resource iterator — Managing resources in long-running transactions.
- Controlling logging
- Traversing a graph
- Domain entities
- Graph algorithm examples
- Unique nodes — Getting or creating a unique node using Cypher and uniqueness constraints.
- Bolt connector — Accessing Neo4j embedded via the Bolt protocol.
- Terminate a transaction — How to terminate (abort) a long-running transaction from another thread.
- Cypher queries — How to use the Cypher query language with Java.
- Query parameters

> When running your own code and Neo4j in the same JVM, there are a few things to keep in mind:
>
> - Do not create or retain more objects than you strictly need to. Large caches in particular tend to promote more objects to the old generation, thus increasing the need for expensive full garbage collections.
> - Do not use internal Neo4j APIs. They are internal to Neo4j and subject to change without notice, which may break or change the behavior of your code.
> - Do not enable the `-XX:+TrustFinalNonStaticFields` JVM flag when running in embedded mode.

Embedding Neo4j in your Java application

After selecting the appropriate edition for your platform, you can embed Neo4j in your Java application by including the Neo4j library JARs in your build. The following sections show how to do this by either altering the build path directly or by using dependency management.

Adding Neo4j to the build path

Get the Neo4j libraries from one of these sources:

- Extract a Neo4j zip/tarball, and use the JAR files found in the *lib/* directory.
- Use the JAR files available from Maven Central Repository.

Add the JAR files to your project:

JDK tools

Append to `-classpath`

Eclipse

- Right-click on the project and then go to *Build Path → Configure Build Path*. In the dialog, select *Add External JARs*, browse the Neo4j *lib/* directory, and select all the JAR files.
- Another option is to use User Libraries.

IntelliJ IDEA

See Libraries, Global Libraries, and the Configure Library dialog.

NetBeans

- Right-click on the *Libraries* node of the project, select *Add JAR/Folder*, browse the Neo4j *lib/* directory and select all the JAR files.
- You can also handle libraries from the project node, see Managing a Project's Classpath.

Editions

The following table outlines the available editions and their names for use with dependency management tools.

> Follow the links in the table for details on dependency configuration with Apache Maven, Apache Buildr, Apache Ivy, Groovy Grape, Grails, and Scala SBT.

Table 2. Neo4j editions

Neo4j Edition	Dependency	Description
Community	org.neo4j:neo4j	A high-performance, fully ACID transactional graph database.
Enterprise	org.neo4j:neo4j-enterprise	Adding advanced monitoring, online backup, and clustering.

Note that the listed dependencies do not contain the implementation, but pull it transitively.

For information regarding licensing, see the Licensing Guide.

Javadocs can be downloaded in JAR files from Maven Central or read in Neo4j Javadocs.

Adding Neo4j as a dependency

You can either go with the top-level artifact or include the individual components directly. The following examples use the top-level artifact approach.

[enterprise-edition]

The examples are only valid for Neo4j Community Edition. To add Neo4j Enterprise Edition as a dependency, please get in contact with Neo4j Professional Services. See Operations Manual → Introduction for details about the Community and Enterprise Editions.

Maven

Add the dependency to your project along the lines of the snippet below. This is usually done in the *pom.xml* file found in the root directory of the project.

```
<project>
...
 <dependencies>
  <dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j</artifactId>
   <version>5.4.0</version>
  </dependency>
  ...
 </dependencies>
...
</project>
```

Where the `artifactId` is found in the Editions table.

Eclipse and Maven

For development in Eclipse, it is recommended to install the m2e plugin and let Maven manage the project build classpath instead. This also adds the possibility to build your project both via the command line with Maven and have a working Eclipse setup for development.

Ivy

Make sure to resolve dependencies from Maven Central. You can use this configuration in your *ivysettings.xml* file:

```
<ivysettings>
  <settings defaultResolver="main"/>
  <resolvers>
    <chain name="main">
      <filesystem name="local">
        <artifact pattern="${ivy.settings.dir}/repository/[artifact]-[revision].[ext]" />
      </filesystem>
      <ibiblio name="maven_central" root="http://repo1.maven.org/maven2/" m2compatible="true"/>
    </chain>
  </resolvers>
</ivysettings>
```

With that in place, add Neo4j by adding the following dependency to your *ivy.xml* file:

```
..
<dependencies>
  ..
  <dependency org="org.neo4j" name="neo4j" rev="5.4.0"/>
  ..
</dependencies>
..
```

The name can be found in the Editions table.

Gradle

An example Gradle build script for including the Neo4j libraries:

```
def neo4jVersion = "5.4.0"
apply plugin: 'java'
repositories {
   mavenCentral()
}
dependencies {
   compile "org.neo4j:neo4j:${neo4jVersion}"
}
```

The coordinates (`org.neo4j:neo4j` in the example) are found in the Editions table.

Starting and stopping

To start the embedded DBMS you instantiate a `org.neo4j.dbms.DatabaseManagementService` and get the `org.neo4j.graphdb.GraphDatabaseService` as follows:

```
managementService = new DatabaseManagementServiceBuilder( databaseDirectory ).build();
graphDb = managementService.database( DEFAULT_DATABASE_NAME );
registerShutdownHook( managementService );
```

If you are using the Enterprise Edition of Neo4j in embedded standalone mode, you have to create your database with the `com.neo4j.dbms.api.EnterpriseDatabaseManagementServiceBuilder` to enable the Enterprise Edition features.

If you are intending to operate embedded clusters, then you should use the `com.neo4j.dbms.api.ClusterDatabaseManagementServiceBuilder` with the appropriate configuration. For maintainability purposes, you can define your embedded DBMS configuration in the neo4j.conf file as follows:

```
dbms.mode=CORE
dbms.default_advertised_address=core01.example.com
dbms.default_listen_address=0.0.0.0
causal_clustering.discovery_type=LIST
causal_clustering.initial_discovery_members=core01.example.com,core02.example.com,core03.example.com
dbms.connector.bolt.enabled=true
dbms.connector.http.enabled=true
```

```
var managementService = new ClusterDatabaseManagementServiceBuilder( homeDirectory )
    .loadPropertiesFromFile( "/path/to/neo4j.conf" )
    .build();
```

It is also possible to use the builder and specify all the parameters programmatically:

```
var defaultAdvertised = new SocketAddress( "core01.example.com" );
var defaultListen = new SocketAddress( "0.0.0.0" );

var initialMembers = List.of(
    new SocketAddress( "core01.example.com" ),
    new SocketAddress( "core02.example.com" ),
    new SocketAddress( "core03.example.com" )
);

var managementService = new ClusterDatabaseManagementServiceBuilder( homeDirectory )
    .setConfig( GraphDatabaseSettings.mode, CORE )
    .setConfig( GraphDatabaseSettings.default_advertised_address, defaultAdvertised )
    .setConfig( GraphDatabaseSettings.default_listen_address, defaultListen )
    .setConfig( CausalClusteringSettings.discovery_type, DiscoveryType.LIST )
    .setConfig( CausalClusteringSettings.initial_discovery_members, initialMembers )
    .setConfig( BoltConnector.enabled, true )
    .setConfig( HttpConnector.enabled, true )
    .build();
```

It is important to carefully consider which services you want to enable, and on which ports and interfaces. If you do not require Bolt or HTTP, then it is better to leave those disabled.

> The `DatabaseManagementService` and `GraphDatabaseService` instances can be shared among multiple threads. Note, however, that you cannot create multiple services pointing to the same database.

To stop the database, call the `shutdown()` method:

```
managementService.shutdown();
```

To make sure Neo4j is shut down properly, add a shutdown hook:

```
private static void registerShutdownHook( final DatabaseManagementService managementService )
{
    // Registers a shutdown hook for the Neo4j instance so that it
    // shuts down nicely when the VM exits (even if you "Ctrl-C" the
    // running application).
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            managementService.shutdown();
        }
    } );
}
```

Starting an embedded database with configuration settings

To start Neo4j with configuration settings, a Neo4j properties file can be loaded as follows:

```
DatabaseManagementService managementService = new DatabaseManagementServiceBuilder( directory )
    .loadPropertiesFromFile( pathToConfig + "neo4j.conf" ).build();
GraphDatabaseService graphDb = managementService.database( DEFAULT_DATABASE_NAME );
```

Configuration settings can also be applied programmatically, as follows:

```
DatabaseManagementService managementService = new DatabaseManagementServiceBuilder( directory)
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )
    .setConfig( GraphDatabaseSettings.transaction_timeout, Duration.ofSeconds( 60 ) )
    .setConfig( GraphDatabaseSettings.preallocate_logical_logs, true ).build();
GraphDatabaseService graphDb = managementService.database( DEFAULT_DATABASE_NAME );
```

Starting an embedded read-only instance

If you want a *read-only view* of the database, create an instance with the `read_only_database_default` configuration setting set to `true`:

```
managementService = new DatabaseManagementServiceBuilder( dir ).setConfig( GraphDatabaseSettings
.read_only_database_default, true ).build();
graphDb = managementService.database( DEFAULT_DATABASE_NAME );
```

The database has to already exist in this case.

> Concurrent access to the same database files by multiple (read-only or write) instances is not supported.

Hello world

A Neo4j graph consists of:

- nodes
- relationships that connect the nodes
- properties on both nodes and relationships

All relationships have a type. For example, if the graph represents a social network, a relationship type could be `KNOWS`. If a relationship of the type `KNOWS` connects two nodes, that is likely to represent two people that know each other. A lot of the semantics of a graph is encoded in the relationship types of the application. Although relationships are directed, they are equally traversed regardless of direction.

For information on project setup, see Embedding Neo4j in your Java application.

> The source code of this example isfound at: EmbeddedNeo4j.java

Preparing the database

Relationship types can be created by using an `enum`. In this example, you only need a single relationship type. This is how to define it:

```
private enum RelTypes implements RelationshipType
{
    KNOWS
}
```

You can also prepare some variables to use:

```
GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;
private DatabaseManagementService managementService;
```

The next step is to start the database server. Note that if the directory given for the database does not already exist, it will be created.

```
managementService = new DatabaseManagementServiceBuilder( databaseDirectory ).build();
graphDb = managementService.database( DEFAULT_DATABASE_NAME );
registerShutdownHook( managementService );
```

Starting a database server is an expensive operation, so do not start up a new instance every time you need to interact with the database. The instance can be shared by multiple threads, and transactions are thread confined.

As seen, you can register a shutdown hook that makes sure the database shuts down when the JVM exits.

```
private static void registerShutdownHook( final DatabaseManagementService managementService )
{
    // Registers a shutdown hook for the Neo4j instance so that it
    // shuts down nicely when the VM exits (even if you "Ctrl-C" the
    // running application).
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            managementService.shutdown();
        }
    } );
}
```

The next step is to interact with the database.

Wrapping operations in a transaction

All operations have to be performed in a transaction. This is a deliberate design decision since transaction demarcation is an important part of working with a real enterprise database. The example below illustrates transaction handling in Neo4j:

```
try ( Transaction tx = graphDb.beginTx() )
{
    // Database operations go here
    tx.commit();
}
```

For more information on transactions, see Transaction management and the Neo4j Javadocs for `org.neo4j.graphdb.Transaction`.

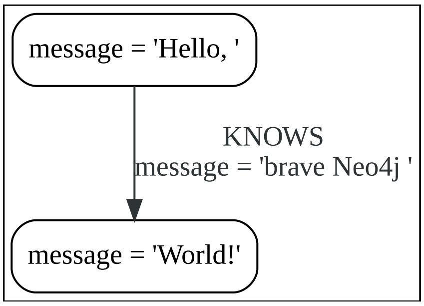
For brevity, the wrapping of operations in a transaction is not spelled out throughout the manual.

Creating a small graph

You can now create a few nodes. This is how to create a small graph consisting of two nodes, connected with one relationship and some properties:

```
firstNode = tx.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = tx.createNode();
secondNode.setProperty( "message", "World!" );

relationship = firstNode.createRelationshipTo( secondNode, RelTypes.KNOWS );
relationship.setProperty( "message", "brave Neo4j " );
```

You now have a graph that looks like this:



Printing the result

After you have created your graph, you can read from it and print the result.

```
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );
```

Which outputs:

```
Hello, brave Neo4j World!
```

Removing the data

In this case, the data is removed before committing:

```
// let's remove the data
firstNode = tx.getNodeById( firstNode.getId() );
secondNode = tx.getNodeById( secondNode.getId() );
firstNode.getSingleRelationship( RelTypes.KNOWS, Direction.OUTGOING ).delete();
firstNode.delete();
secondNode.delete();
```

Deleting a node that still has relationships when the transaction commits will fail. This is to make sure relationships always have a start node and an end node.

Shutting down the database server

Finally, shut down the database server *when the application finishes:*

```
managementService.shutdown();
```

Property values

This describes how both nodes and relationships can have properties.

Properties are named values where the name is a string. Property values can be either a primitive or an array of one primitive type. For example `String`, `int`, and `int[]` values are valid for properties.

> `NULL` is **not a valid** property value. Setting a property to `NULL` is equivalent to deleting the property.

Table 3. Property value types

Type	Description
`boolean`	
`byte`	8-bit integer.
`short`	16-bit integer.
`int`	32-bit integer.
`long`	64-bit integer.
`float`	32-bit IEEE 754 floating-point number.
`double`	64-bit IEEE 754 floating-point number.
`char`	16-bit unsigned integers representing Unicode characters.
`String`	Sequence of Unicode characters.
`org.neo4j.graphdb.spatial.Point`	A 2D or 3D point object in a given coordinate system.
`java.time.LocalDate`	An instant capturing of the date, but not the time and timezone.
`java.time.OffsetTime`	An instant capturing of the time of day and the timezone offset, but not the date.
`java.time.LocalTime`	An instant capturing of the time of day, but not the date and timezone.
`java.time.ZonedDateTime`	An instant capturing of the date, time, and timezone.
`java.time.LocalDateTime`	An instant capturing of the date and time, but not the timezone.
`java.time.temporal.TemporalAmount`	A temporal amount. This captures the difference in time between two instants.

For further details on float/double values, see Java Language Specification.

Note that there are two cases where more than one Java type is mapped to a single Cypher type. When this happens, type information is lost. If these objects are returned from procedures, the original types cannot be recreated:

- A Cypher `Duration` is created when either `java.time.Duration` or `java.time.Period` is provided. If `Duration` is returned, only the common interface `java.time.temporal.TemporalAmount` remains.
- A Cypher `DateTime` is created when `java.time.OffsetDateTime` is provided. If `DateTime` is returned, it is converted into `java.time.ZonedDateTime`.

Strings that contain special characters can have inconsistent or non-deterministic ordering in Neo4j. For details, see Cypher Manual → Sorting of special characters.

Using indexes

It is possible to create and use all the index types described in Cypher Manual → Indexes.

This section demonstrates how to work with indexes with an example of a user database. For information about how to create an index on all `User` nodes that have a `username` property, see Cypher Manual → Create a single-property index for nodes.

The source code used in this example is found at: EmbeddedNeo4jWithIndexing.java

Begin with starting the database server:

```
DatabaseManagementService managementService = new DatabaseManagementServiceBuilder( databaseDirectory
).build();
GraphDatabaseService graphDb = managementService.database( DEFAULT_DATABASE_NAME );
```

Then, you can configure the database to index users by name. This only needs to be done once.

Note that schema changes and data changes are not allowed in the same transaction. Each transaction must either change the schema or the data, but not both.

```
IndexDefinition usernamesIndex;
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = tx.schema();
    usernamesIndex = schema.indexFor( Label.label( "User" ) )  ①
            .on( "username" )                                 ②
            .withName( "usernames" )                          ③
            .create();                                        ④
    tx.commit();                                              ⑤
}
```

① A single-property index is defined on a label in combination with a property name. Start your index definition by specifying the node label.

② Next, define the property that should be part of this index. Index all nodes with the `User` label, that also have a `username` property. This way, you can find `User` nodes by their `username` properties.

③ An index always has a name. If not specified, it will be generated for you.

④ Calling `create` is necessary for the index definition to be created in the database. This index is now created, but it still only exists in your current transaction.

⑤ Committing the transaction commits your new index to the database. It will become available for use once it has finished populating itself with the existing data in your database.

Indexes are populated asynchronously when they are first created. It is possible to use the core API to wait for the index population to complete:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = tx.schema();
    schema.awaitIndexOnline( usernamesIndex, 10, TimeUnit.SECONDS );
}
```

It is also possible to query the progress of the index population:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = tx.schema();
    System.out.println( String.format( "Percent complete: %1.0f%%",
        schema.getIndexPopulationProgress( usernamesIndex ).getCompletedPercentage() ) );
}
```

Now you can add the users:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );

    // Create some users
    for ( int id = 0; id < 100; id++ )
    {
        Node userNode = tx.createNode( label );
        userNode.setProperty( "username", "user" + id + "@neo4j.org" );
    }
    System.out.println( "Users created" );
    tx.commit();
}
```

For information on how to properly close `ResourceIterators` returned from index lookups, read Resource iterator.

And this is how to find a user by ID:

```
Label label = Label.label( "User" );
int idToFind = 45;
String nameToFind = "user" + idToFind + "@neo4j.org";
try ( Transaction tx = graphDb.beginTx() )
{
    try ( ResourceIterator<Node> users =
                  tx.findNodes( label, "username", nameToFind ) )
    {
        ArrayList<Node> userNodes = new ArrayList<>();
        while ( users.hasNext() )
        {
            userNodes.add( users.next() );
        }

        for ( Node node : userNodes )
        {
            System.out.println(
                    "The username of user " + idToFind + " is " + node.getProperty( "username" ) );
        }
    }
}
```

When updating the name of a user, the index is updated as well:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );
    int idToFind = 45;
    String nameToFind = "user" + idToFind + "@neo4j.org";

    for ( Node node : loop( tx.findNodes( label, "username", nameToFind ) ) )
    {
        node.setProperty( "username", "user" + (idToFind + 1) + "@neo4j.org" );
    }
    tx.commit();
}
```

When deleting a user, it is automatically removed from the index:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );
    int idToFind = 46;
    String nameToFind = "user" + idToFind + "@neo4j.org";

    for ( Node node : loop( tx.findNodes( label, "username", nameToFind ) ) )
    {
        node.delete();
    }
    tx.commit();
}
```

In case you change your data model, you can drop the index as well:

```
try ( Transaction tx = graphDb.beginTx() )
{
    IndexDefinition usernamesIndex = tx.schema().getIndexByName( "usernames" ); ①
    usernamesIndex.drop();
    tx.commit();
}
```

① You look up the index by the index name you gave it when you created it. Index names are guaranteed to be unique, to ensure that you will not mistakenly find and drop the wrong index.

Resource iterator

Inside a long-running transaction, it is a good practice to ensure that any `org.neo4j.graphdb.ResourceIterator` obtained inside the transaction are closed as early as possible. This is either achieved by exhausting the iterator or by explicitly calling its close method.

The following is an example of how to work with `ResourceIterator`. If you do not exhaust the iterator, you can close it explicitly using the `close()` method.

```
Label label = Label.label( "User" );
int idToFind = 45;
String nameToFind = "user" + idToFind + "@neo4j.org";
try ( Transaction tx = graphDb.beginTx();
      ResourceIterator<Node> users = tx.findNodes( label, "username", nameToFind ) )
{
    Node firstUserNode;
    if ( users.hasNext() )
    {
        firstUserNode = users.next();
    }
    users.close();
    // ... Do stuff with the firstUserNode we found ...
}
```

Controlling logging

Neo4j embedded provides logging via its own `org.neo4j.logging.Log` layer and does not natively use any existing Java logging framework. All logging events produced by Neo4j have a name, a level, and a message. The name is a fully qualified class name (FQCN).

Table 4. Neo4j uses the following log levels:

Level Name	Description
`ERROR`	For serious errors that are almost always fatal.
`WARN`	For events that are serious, but not fatal.
`INFO`	Informational events.
`DEBUG`	Debugging events.

To enable logging, an implementation of `org.neo4j.logging.LogProvider` must be provided to the `org.neo4j.dbms.api.DatabaseManagementServiceBuilder`, as follows:

```
LogProvider logProvider = new MyCustomLogProvider( output );
managementService = new DatabaseManagementServiceBuilder( databaseDirectory ).setUserLogProvider(
logProvider ).build();
```

Traversing a graph

This page describes how to traverse a graph using the Traversal Framework.

The Matrix example

This example of a traversed graph uses some of the characters featured in the film trilogy "The Matrix". The source code can be found in NewMatrix.java.

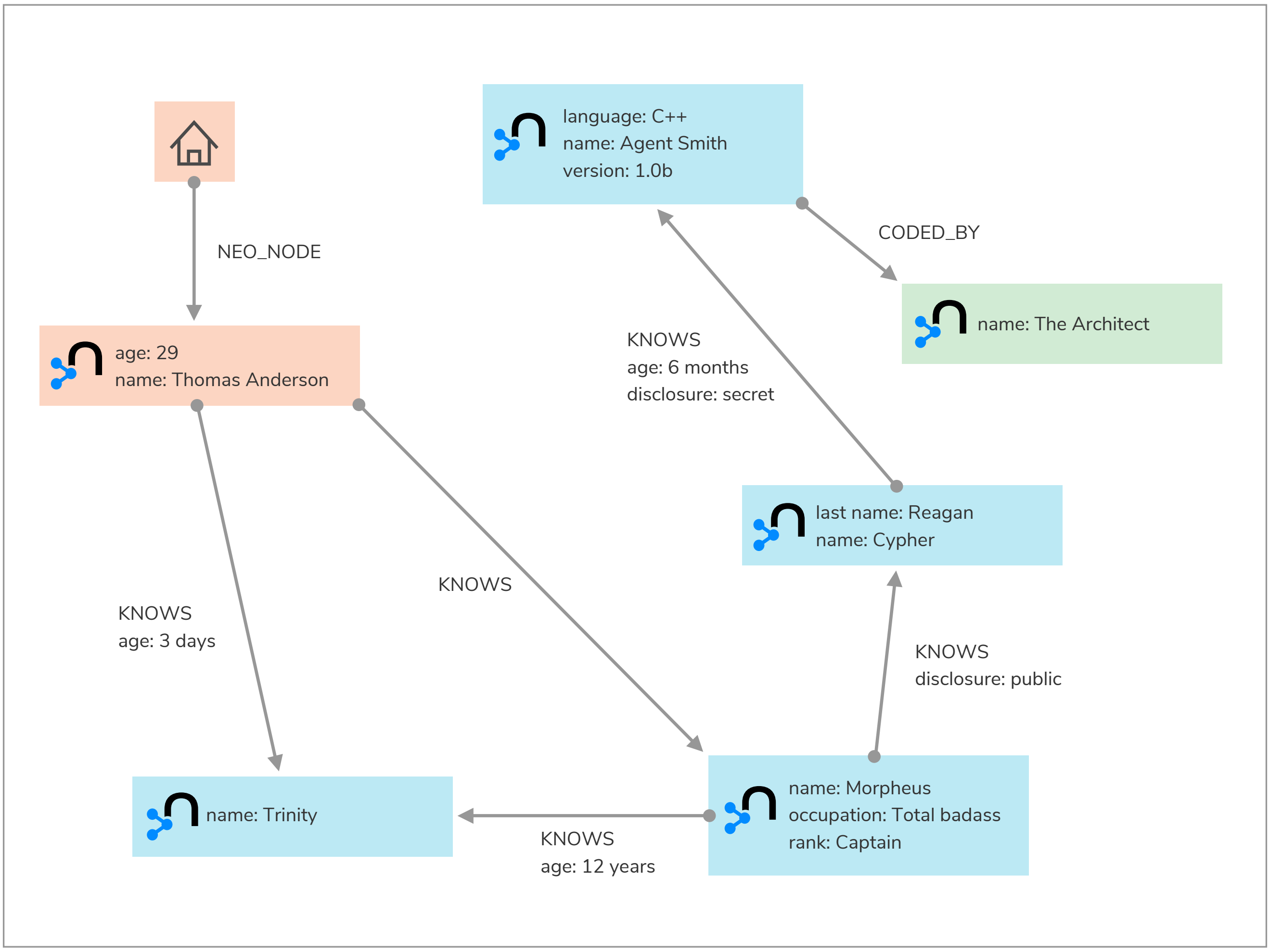


Figure 1. Matrix node space view

The following is an example of how to define the `TraversalDescription` to find all the friends and friends of friends of a person X, and how to return the results excluding the person X:

```
private Traverser getFriends( Transaction transaction, final Node person )
{
    TraversalDescription td = transaction.traversalDescription()
            .breadthFirst()
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator( Evaluators.excludeStartPosition() );
    return td.traverse( person );
}
```

Perform the actual traversal and print the names of all found friends:

```
int numberOfFriends = 0;
String output = neoNode.getProperty( "name" ) + "'s friends:\n";
Traverser friendsTraverser = getFriends( tx, neoNode );
for ( Path friendPath : friendsTraverser )
{
    output += "At depth " + friendPath.length() + " => "
              + friendPath.endNode()
                      .getProperty( "name" ) + "\n";
    numberOfFriends++;
}
output += "Number of friends found: " + numberOfFriends + "\n";
```

The traversal returns the following output:

```
Thomas Anderson's friends:
At depth 1 => Morpheus
At depth 1 => Trinity
At depth 2 => Cypher
At depth 3 => Agent Smith
Number of friends found: 4
```

Solving the Matrix

To find the mastermind behind "The Matrix", follow all outgoing relationships that have the type KNOWS or CODED_BY. The final node connected to the relationship CODED_BY will be the Matrix's coder.

```
private Traverser findHackers( Transaction transaction, final Node startNode )
{
    TraversalDescription td = transaction.traversalDescription()
        .breadthFirst()
        .relationships( RelTypes.CODED_BY, Direction.OUTGOING )
        .relationships( RelTypes.KNOWS, Direction.OUTGOING )
        .evaluator(Evaluators.includeWhereLastRelationshipTypeIs( RelTypes.CODED_BY ) );
    return td.traverse( startNode );
}
```

Here is how to print out the result and find all hackers:

```
String output = "Hackers:\n";
int numberOfHackers = 0;
Traverser traverser = findHackers( tx, getNeoNode( tx ) );
for ( Path hackerPath : traverser )
{
    output += "At depth " + hackerPath.length() + " => " + hackerPath.endNode().getProperty( "name" ) +
"\n";
    numberOfHackers++;
}
output += "Number of hackers found: " + numberOfHackers + "\n";
```

Now you know who coded the Matrix:

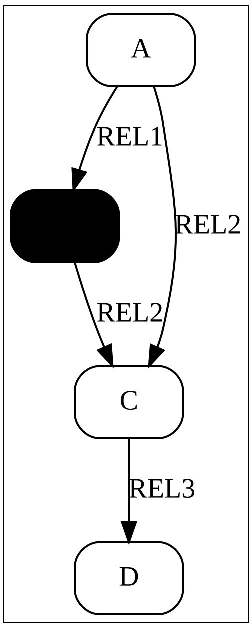
```
Hackers:
At depth 4 => The Architect
Number of hackers found: 1
```

Walking an ordered path

The following example shows how to use a custom Evaluator to find paths that adhere to a predefined order. The source code can be found in OrderedPath.java.

First, you create the example graph that will be traversed over:

```
Node A = tx.createNode();
Node B = tx.createNode();
Node C = tx.createNode();
Node D = tx.createNode();

A.createRelationshipTo( C, REL2 );
C.createRelationshipTo( D, REL3 );
A.createRelationshipTo( B, REL1 );
B.createRelationshipTo( C, REL2 );
```



When setting up the traversal of the graph, store the order of the relationships (REL1 → REL2 → REL3) in an ArrayList. Upon traversal, the Evaluator can check against it to ensure that the only paths included and returned have the predefined order of relationships:

```
final ArrayList<RelationshipType> orderedPathContext = new ArrayList<>();
orderedPathContext.add( REL1 );
orderedPathContext.add( REL2 );
orderedPathContext.add( REL3 );
TraversalDescription td = tx.traversalDescription()
    .evaluator( new Evaluator()
    {
        @Override
        public Evaluation evaluate( final Path path )
        {
            if ( path.length() == 0 )
            {
                return Evaluation.EXCLUDE_AND_CONTINUE;
            }
            RelationshipType expectedType = orderedPathContext.get( path.length() - 1 );
            boolean isExpectedType = path.lastRelationship()
                    .isType( expectedType );
            boolean included = path.length() == orderedPathContext.size() && isExpectedType;
            boolean continued = path.length() < orderedPathContext.size() && isExpectedType;
            return Evaluation.of( included, continued );
        }
    } )
    .uniqueness( Uniqueness.NODE_PATH ); ①
```

Note that the uniqueness is set to Uniqueness.NODE_PATH. This makes it possible to revisit the same node during the traversal, but not the same path.

Then, you can perform the traversal and print all ordered paths found:

```
Traverser traverser = td.traverse( tx.getNodeById( A.getId() ) );
PathPrinter pathPrinter = new PathPrinter( "name" );
for ( Path path : traverser )
{
    output += Paths.pathToString( path, pathPrinter );
}
```

This returns the following output:

```
(A)--[REL1]-->(B)--[REL2]-->(C)--[REL3]-->(D)
```

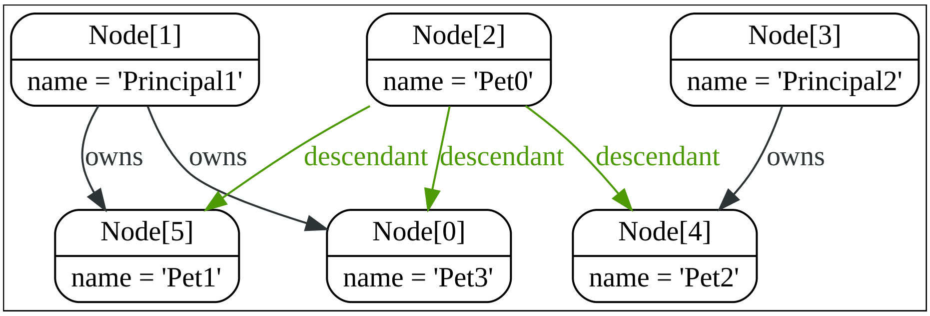
In this case, a customized class is used to format the path output.

```
static class PathPrinter implements Paths.PathDescriptor<Path>
{
    private final String nodePropertyKey;

    public PathPrinter( String nodePropertyKey )
    {
        this.nodePropertyKey = nodePropertyKey;
    }

    @Override
    public String nodeRepresentation( Path path, Node node )
    {
        return "(" + node.getProperty( nodePropertyKey, "" ) + ")";
    }

    @Override
    public String relationshipRepresentation( Path path, Node from, Relationship relationship )
    {
        String prefix = "--", suffix = "--";
        if ( from.equals( relationship.getEndNode() ) )
        {
            prefix = "<--";
        }
        else
        {
            suffix = "-->";
        }
        return prefix + "[" + relationship.getType().name() + "]" + suffix;
    }
}
```

Uniqueness of Paths in traversals

The following example demonstrates the use of node uniqueness. It lists all pets descended from other pets that are owned by `Principal1`:

Descendants example graph



To return all descendants of Pet0 which are owned by Principal1 (i.e. Pet1 and Pet3), the uniqueness of the traversal needs to be set to NODE_PATH rather than the default NODE_GLOBAL. This way, nodes can be traversed more than once, and paths that have different nodes with some in common (like the start and the end nodes) can be returned.

```
Node dataTarget = data.get().get( "Principal1" );
String output = "";
int count = 0;
try ( Transaction transaction = graphdb().beginTx() )
{
    start = transaction.getNodeById( start.getId() );
    final Node target = transaction.getNodeById( dataTarget.getId() );
    TraversalDescription td = transaction.traversalDescription()
            .uniqueness( Uniqueness.NODE_PATH )
            .evaluator( new Evaluator()
    {
        @Override
        public Evaluation evaluate( Path path )
        {
            boolean endNodeIsTarget = path.endNode().equals( target );
            return Evaluation.of( endNodeIsTarget, !endNodeIsTarget );
        }
    } );

    Traverser results = td.traverse( start );
}
```

This will return the following paths:

```
(2)-[descendant,2]->(0)<-[owns,5]-(1)
(2)-[descendant,0]->(5)<-[owns,3]-(1)
```

You can also see how this differs from using NODE_GLOBAL uniqueness. Note that the TraversalDescription object is immutable, so a new instance is required.

```
TraversalDescription nodeGlobalTd = tx.traversalDescription().uniqueness( Uniqueness.NODE_PATH ).
evaluator( new Evaluator()
{
    @Override
    public Evaluation evaluate( Path path )
    {
        boolean endNodeIsTarget = path.endNode().equals( target );
        return Evaluation.of( endNodeIsTarget, !endNodeIsTarget );
    }
} ).uniqueness( Uniqueness.NODE_GLOBAL );
Traverser results = nodeGlobalTd.traverse( start );
```

Now only one path is returned because the node Principal1 can only be traversed once:

```
(2)-[descendant,2]->(0)<-[owns,5]-(1)
```

Domain entities

Domain entities can be wrapped around a node. The same approach can be used with relationships.

> The source code of the examples isfound at: Person.java

First off, store the node and make it accessible inside the package:

```java
private final Node underlyingNode;

Person( GraphDatabaseService databaseService, Transaction transaction, Node personNode )
{
    this.databaseService = databaseService;
    this.transaction = transaction;
    this.underlyingNode = personNode;
}

protected Node getUnderlyingNode()
{
    return underlyingNode;
}
```

Delegate attributes to the node:

```java
public String getName()
{
    return (String)underlyingNode.getProperty( NAME );
}
```

Make sure to override these methods:

```java
@Override
public int hashCode()
{
    return underlyingNode.hashCode();
}

@Override
public boolean equals( Object o )
{
    return o instanceof Person &&
            underlyingNode.equals( ( (Person)o ).getUnderlyingNode() );
}

@Override
public String toString()
{
    return "Person[" + getName() + "]";
}
```

Graph algorithm examples

For details on the graph algorithm usage, see the Neo4j Javadocs for `org.neo4j.graphalgo.GraphAlgoFactory`.

	The source code used in the example isfound at: PathFindingDocTest.java

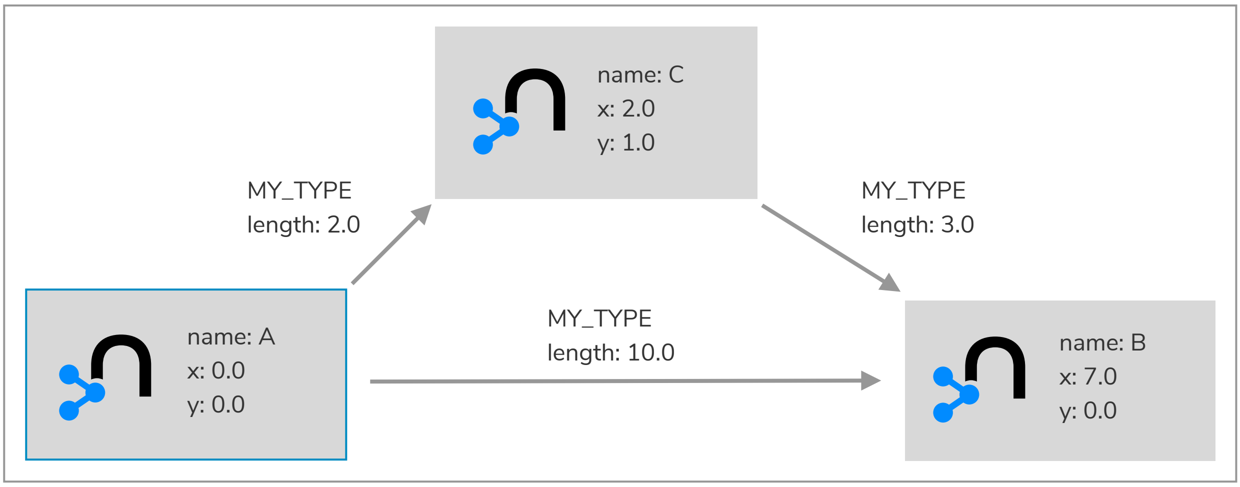
Calculating the shortest path (least number of relationships) between two nodes:

```
Node startNode = tx.createNode();
Node middleNode1 = tx.createNode();
Node middleNode2 = tx.createNode();
Node middleNode3 = tx.createNode();
Node endNode = tx.createNode();
createRelationshipsBetween( startNode, middleNode1, endNode );
createRelationshipsBetween( startNode, middleNode2, middleNode3, endNode );

// Will find the shortest path between startNode and endNode via
// "MY_TYPE" relationships (in OUTGOING direction), like f.ex:
//
// (startNode)-->(middleNode1)-->(endNode)
//
PathFinder<Path> finder = GraphAlgoFactory.shortestPath( new BasicEvaluationContext( tx, graphDb ),
    PathExpanders.forTypeAndDirection( ExampleTypes.MY_TYPE, Direction.OUTGOING ), 15 );
Iterable<Path> paths = finder.findAllPaths( startNode, endNode );
```

Using the Dijkstra Source-Target Shortest Path, you can calculate the cheapest path between node A and B, where each relationship has weight (e.g., cost) and the path(s) with least cost are found.

```
PathFinder<WeightedPath> finder = GraphAlgoFactory.dijkstra( new BasicEvaluationContext( tx, graphDb ),
    PathExpanders.forTypeAndDirection( ExampleTypes.MY_TYPE, Direction.BOTH ), "cost" );

WeightedPath path = finder.findSinglePath( nodeA, nodeB );

// Get the weight for the found path
path.weight();
```

Using A* Shortest Path, you can calculate the cheapest path between node A and B, where the "cheapest" means, for example, the path in a network of roads, which has the shortest length between node A and B. This is the example graph:



```
Node nodeA = createNode( "name", "A", "x", 0d, "y", 0d );
Node nodeB = createNode( "name", "B", "x", 7d, "y", 0d );
Node nodeC = createNode( "name", "C", "x", 2d, "y", 1d );
Relationship relAB = createRelationship( nodeA, nodeC, "length", 2d );
Relationship relBC = createRelationship( nodeC, nodeB, "length", 3d );
Relationship relAC = createRelationship( nodeA, nodeB, "length", 10d );

EstimateEvaluator<Double> estimateEvaluator = ( node, goal ) ->
{
    double dx = (Double) node.getProperty( "x" ) - (Double) goal.getProperty( "x" );
    double dy = (Double) node.getProperty( "y" ) - (Double) goal.getProperty( "y" );
    return Math.sqrt( Math.pow( dx, 2 ) + Math.pow( dy, 2 ) );
};
PathFinder<WeightedPath> astar = GraphAlgoFactory.aStar( new BasicEvaluationContext( tx, graphDb ),
        PathExpanders.allTypesAndDirections(),
        CommonEvaluators.doubleCostEvaluator( "length" ), estimateEvaluator );
WeightedPath path = astar.findSinglePath( nodeA, nodeB );
```

Unique nodes

This describes how to ensure the uniqueness of a property when creating nodes.

For an overview of unique nodes, see Transaction management → Creating unique nodes.

The source code for the examples can be found in: GetOrCreateDocIT.java

Create a unique constraint:

```
try ( Transaction tx = graphdb.beginTx() )
{
    tx.schema()
        .constraintFor( Label.label( "User" ) )
        .assertPropertyIsUnique( "name" )
        .withName( "usernames" )
        .create();
    tx.commit();
}
```

Use `MERGE` to create a unique node:

```
Node result = null;
ResourceIterator<Node> resultIterator = null;
try ( Transaction tx = graphDb.beginTx() )
{
    String queryString = "MERGE (n:User {name: $name}) RETURN n";
    Map<String, Object> parameters = new HashMap<>();
    parameters.put( "name", username );
    resultIterator = tx.execute( queryString, parameters ).columnAs( "n" );
    result = resultIterator.next();
    tx.commit();
    return result;
}
```

The `MERGE` clause takes either a read-lock on the matching node that already exists or a write-lock and ensures that the current transaction is the only one creating the node.

It is technically possible to use a *lock node* or a *lock property* but this should be avoided if possible. Using the *lock node* pattern is difficult to do correctly and it has worse performance characteristics because it always involves a write-lock.

You might also be tempted to use Java synchronization for pessimistic locking, but this is dangerous. By mixing locks in Neo4j and the Java runtime, it is possible to produce deadlocks that are not detectable by Neo4j. As long as all locking is done by Neo4j, all deadlocks will be detected and avoided.

Bolt connector

This describes how to open a Bolt connector to your embedded instance to get GUI administration and other benefits. Accessing Neo4j embedded via the Bolt protocol.

The Neo4j Browser and the official Neo4j Drivers use the Bolt database protocol to communicate with Neo4j. By default, Neo4j embedded does not expose a Bolt connector, but you can enable one. Doing so allows you to connect the services of Neo4j Browser to your embedded instance.

It also gives you a way to incrementally transfer an existing Embedded application to use Neo4j Drivers instead. Migrating to Neo4j Drivers means you can run Neo4j embedded or Neo4j server, without having to change your application code.

To add a Bolt Connector to your embedded database, you must add the Bolt extension to your classpath. This is done by adding a dependency to your project:

```
<project>
...
 <dependencies>

  <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-bolt</artifactId>
    <version>5.4.0</version>
  </dependency>
  ...
 </dependencies>
...
</project>
```

With this dependency in place, you can configure Neo4j to enable the Bolt connector:

The source code for the example can befound at: EmbeddedNeo4jWithBolt.java

```
DatabaseManagementService managementService = new DatabaseManagementServiceBuilder( DB_PATH )
    .setConfig( BoltConnector.enabled, true )
    .setConfig( BoltConnector.listen_address, new SocketAddress( "localhost", 7687 ) )
    .build();
```

Terminate a transaction

You can terminate (abort) a long-running transaction from another thread.

The source code for the examples can befound at: TerminateTransactions.java

First, start the database server:

```
DatabaseManagementService managementService = new DatabaseManagementServiceBuilder( databaseDirectory
).build();
GraphDatabaseService graphDb = managementService.database( DEFAULT_DATABASE_NAME );
```

Then, start creating an infinite binary tree of nodes in the database as an example of a long-running transaction:

```
RelationshipType relType = RelationshipType.withName( "CHILD" );
Queue<Node> nodes = new LinkedList<>();
int depth = 1;

try ( Transaction tx = graphDb.beginTx() )
{
    Node rootNode = tx.createNode();
    nodes.add( rootNode );

    for (; true; depth++) {
        int nodesToExpand = nodes.size();
        for (int i = 0; i < nodesToExpand; ++i) {
            Node parent = nodes.remove();

            Node left = tx.createNode();
            Node right = tx.createNode();

            parent.createRelationshipTo( left, relType );
            parent.createRelationshipTo( right, relType );

            nodes.add( left );
            nodes.add( right );
        }
    }
}
catch ( TransactionTerminatedException ignored )
{
    return String.format( "Created tree up to depth %s in 1 sec", depth );
}
```

After waiting for some time, terminate the transaction from a separate thread:

```
tx.terminate();
```

Running this executes the long-running transaction for about one second and prints the maximum depth of the tree that had been created before the transaction was terminated. No changes have been made to the data (because of the transaction’s termination) and the result is as if no operations have been performed.

This is an example output:

```
Created tree up to depth 18 in 1 sec
```

Finally, the database can be shut down again:

```
managementService.shutdown();
```

Cypher queries

In Java, you can use the Cypher query language as per the example below.

The source code for the examples can befound at: JavaQuery.java

First, you can add some data:

```
DatabaseManagementService managementService = new DatabaseManagementServiceBuilder( databaseDirectory
).build();
GraphDatabaseService db = managementService.database( DEFAULT_DATABASE_NAME );

try ( Transaction tx = db.beginTx())
{
    Node myNode = tx.createNode();
    myNode.setProperty( "name", "my node" );
    tx.commit();
}
```

Execute a query:

```
try ( Transaction tx = db.beginTx();
      Result result = tx.execute( "MATCH (n {name: 'my node'}) RETURN n, n.name" ) )
{
    while ( result.hasNext() )
    {
        Map<String,Object> row = result.next();
        for ( Entry<String,Object> column : row.entrySet() )
        {
            rows += column.getKey() + ": " + column.getValue() + "; ";
        }
        rows += "\n";
    }
}
```

In this example, you can also see how to iterate over the rows of the `org.neo4j.graphdb.Result`.

The code will generate:

```
n: Node[0]; n.name: my node;
```

When using the `Result` object, you should consume the entire result (iterate over all rows using `next()`, by iterating over the iterator from `columnAs()` or calling for example `resultAsString()`). Failing to do so will not properly clean up resources used by the `Result` object, leading to unwanted behavior, such as leaking transactions. In case you do not want to iterate over all of the results, make sure you invoke `close()` as soon as you are done, to release the resources tied to the result.

The recommended way to handle results is to use a try-with-resources statement. This ensures that the result is closed at the end of the statement.

You can also get a list of the columns in the result:

```
List<String> columns = result.columns();
```

This gives you:

```
[n, n.name]
```

Use the following to fetch the result items from a single column. In this case, you must read the property from the node, and not from the result:

```
Iterator<Node> n_column = result.columnAs( "n" );
n_column.forEachRemaining( node -> nodeResult = node + ": " + node.getProperty( "name" ) );
```

In this case, there is only one node in the result:

```
Node[0]: my node
```

Use this only if the result contains a single column or you are interested in a single column of the result.

`resultAsString()`, `writeAsStringTo()`, `columnAs()` cannot be called more than once on the same `Result` object, as they consume the result. In the same way, part of the result gets consumed for every call to `next()`. You should instead use only one and if you need the facilities of the other methods on the same query result instead create a new `Result`.

For more information on the Java interface to Cypher, see the Neo4j Javadocs.

For more information and examples for Cypher, see Neo4j Cypher Manual.

Query parameters

The following examples illustrate how to use parameters when executing Cypher queries from Java.

For more information on parameters, see the Neo4j Cypher Manual.

Node ID:

```
Map<String,Object> params = new HashMap<>();
params.put( "id", 0 );

String query =
    "MATCH (n)" + "\n" +
    "WHERE id(n) = $id" + "\n" +
    "RETURN n.name";

Result result = transaction.execute( query, params );
```

Node object:

```
Map<String,Object> params = new HashMap<>();
params.put( "node", bobNode );

String query =
    "MATCH (n:Person)" + "\n" +
    "WHERE n = $node" + "\n" +
    "RETURN n.name";

Result result = transaction.execute( query, params );
```

Multiple node IDs:

```
Map<String,Object> params = new HashMap<>();
params.put( "ids", asList( 0, 1, 2 ) );

String query =
    "MATCH (n)" + "\n" +
    "WHERE id(n) IN $ids" + "\n" +
    "RETURN n.name";

Result result = transaction.execute( query, params );
```

String literal:

```
Map<String,Object> params = new HashMap<>();
params.put( "name", "Johan" );

String query =
    "MATCH (n:Person)" + "\n" +
    "WHERE n.name = $name" + "\n" +
    "RETURN n";

Result result = transaction.execute( query, params );
```

Numeric parameters for `SKIP` and `LIMIT`:

```
Map<String,Object> params = new HashMap<>();
params.put( "s", 1 );
params.put( "l", 1 );

String query =
    "MATCH (n:Person)" + "\n" +
    "RETURN n.name" + "\n" +
    "SKIP $s" + "\n" +
    "LIMIT $l";

Result result = transaction.execute( query, params );
```

Regular expression:

```
Map<String,Object> params = new HashMap<>();
params.put( "regex", ".*h.*" );

String query =
    "MATCH (n:Person)" + "\n" +
    "WHERE n.name =~ $regex" + "\n" +
    "RETURN n.name";

Result result = transaction.execute( query, params );
```

Create a node with properties:

```
Map<String,Object> props = new HashMap<>();
props.put( "name", "Andy" );
props.put( "position", "Developer" );

Map<String,Object> params = new HashMap<>();
params.put( "props", props );

String query = "CREATE ($props)";

transaction.execute( query, params );
```

Create multiple nodes with properties:

```
Map<String,Object> n1 = new HashMap<>();
n1.put( "name", "Andy" );
n1.put( "position", "Developer" );
n1.put( "awesome", true );

Map<String,Object> n2 = new HashMap<>();
n2.put( "name", "Michael" );
n2.put( "position", "Developer" );
n2.put( "children", 3 );

Map<String,Object> params = new HashMap<>();
List<Map<String,Object>> maps = asList( n1, n2 );
params.put( "props", maps );

String query =
    "UNWIND $props AS properties" + "\n" +
    "CREATE (n:Person)" + "\n" +
    "SET n = properties" + "\n" +
    "RETURN n";

Result result = transaction.execute( query, params );
```

Setting all properties on a node:

```
Map<String, Object> n1 = new HashMap<>();
n1.put( "name", "Andy" );
n1.put( "position", "Developer" );

Map<String, Object> params = new HashMap<>();
params.put( "props", n1 );

String query =
    "MATCH (n:Person)" + "\n" +
    "WHERE n.name = 'Michaela'" + "\n" +
    "SET n = $props";

transaction.execute( query, params );
```

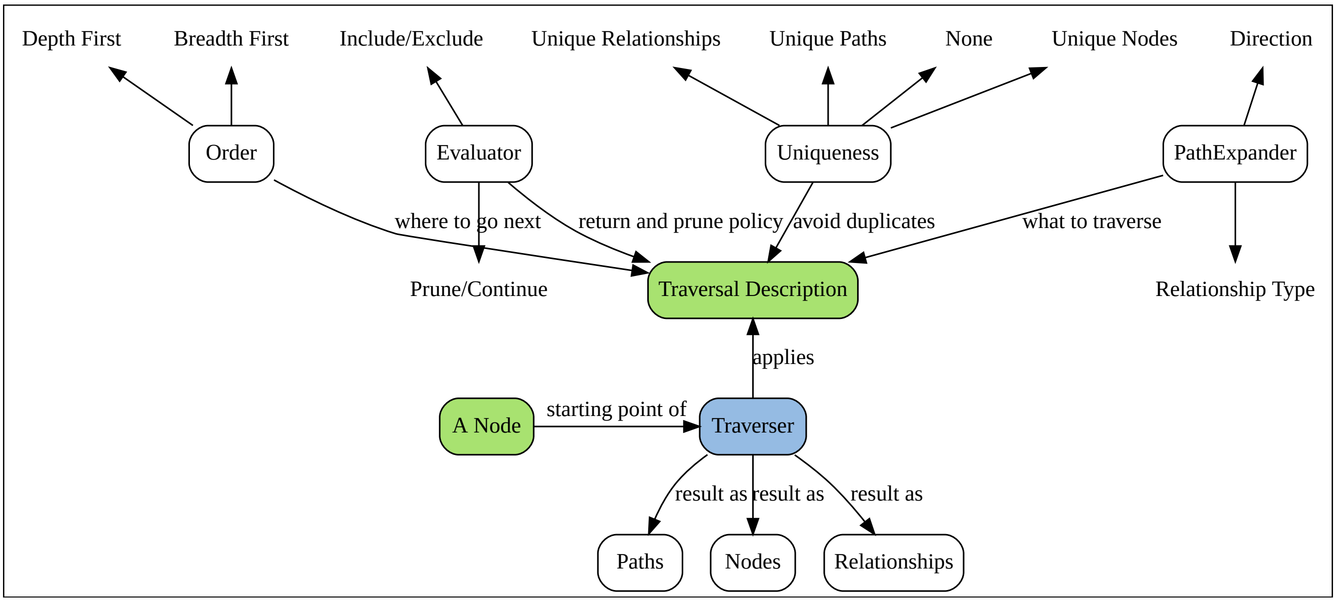
Traversal Framework

The Neo4j Traversal Framework Java API is a callback-based, lazily-executed way of specifying desired movements through a graph in Java. Some traversal examples can be found in Traversing a graph.

Main concepts

A traversal takes a start node on a graph and returns a set of paths representing the visited nodes and their relationships. Traversals are defined by a traversal description, which may contain the following elements:

- *Starting node* — defines where the traversal begins.
- *Pathexpander* — defines what to traverse, typically in terms of relationship direction and type.
- *Uniqueness* — defines the restrictions of previously traversed nodes and relationships on the graph.
- *Evaluator* — decides what to return and whether to stop or continue the traversal beyond its current position.
- *Order* — defines in which order paths are expanded, for example, `depth-first` or `breadth-first`.



Using the Traversal Framework

The Traversal Framework can be used embedded in Java applications. It can also be used when extending Neo4j with a User-defined Procedure. For an example, see User-defined procedure with a Traversal Framework.

Traversal Framework vs Cypher

Although the Traversal Framework is less readable and more complex than the Cypher query language, it offers a powerful approach to traversing the graph. This is because the Traversal Framework can dynamically make custom choices at each step of the traversal, thus making the process more expressive and potentially more performant than Cypher.

Some of the advantages of using the Traversal Framework over Cypher include:

- The Traversal Framework allows the use of any desired Java library to help in the evaluation of the traversal.
- It allows customized pruning during the traversal of a path, which could potentially improve the performance of a traversal. See Evaluator for more information.
- With Cypher, it is not possible to specify the order in which paths are expanded (e.g. depth-first). However, with the Traversal Framework, it is possible to specify the order of paths traversed.
- With Cypher, relationships are only traversed when `RELATIONSHIP_GLOBAL` uniqueness is specified. By using the Traversal Framework, it is possible to specify the uniqueness constraints on the path traversed.

> ⚠ It is generally recommended to use Cypher wherever it is possible. However, when using the Traversal Framework, keep in mind that:
>
> - Cypher uses memory tracking that allows a query to be aborted if it occupies too much memory. However, when mixing Cypher with Java in the Traversal API (e.g. on a function), you may run out of memory.
> - Do not reuse objects fetched during a traversal in another transaction. Instead, use their IDs to fetch new ones.

Traversal Framework Java API

`TraversalDescription`

`TraversalDescription` is the main interface used for defining and initializing traversals. It is not meant to be implemented by users of the Traversal Framework, but it works as a means for the user to describe traversals.

`TraversalDescription` instances are immutable. Their methods return a new `TraversalDescription` that is modified according to the object that the method was invoked on, with its arguments. The method `traverse()` returns the result of the traversal defined in the `TraversalDescription`.

The `Traverser` object

The `Traverser` object is the result of invoking `traverse()` on `TraversalDescription`. It represents a traversal positioned on the graph and a specification of the format of the result. The actual traversal is performed lazily each time the `next()` method of the iterator of the `Traverser` is invoked.

The following is an example of `Traverser` using default values, that is, Uniqueness: `NODE_GLOBAL`, Expander: `BOTH`, and Branch Ordering: `PREORDER_DEPTH_FIRST`:

```java
TraversalDescription td;
try ( Transaction tx = graphDb.beginTx() ) {
     td = tx.traversalDescription();
}

Traverser traverser = td.traverse( startNode );

for ( Path path : traverser ) {
    // Extend as needed
}
```

Additionally, the `Traverser` features methods for reading the last `relationships()` and `nodes()` for each of the returned `Paths` and `metadata()`. It has convenience methods for finding the total number of returned paths, `getNumberOfPathsReturned()`, and the number of relationships traversed in `getNumberOfRelationshipsTraversed()`.

Using `relationships()`

The method `relationships()` defines relationship types and, optionally, the relationship direction to be traversed. By default, all relationships are traversed, regardless of their type. However, if one or more relationships are added, then only the added types are traversed.

```java
TraversalDescription td = transaction.traversalDescription()
    .relationships(RelationshipType.withName("A"))
    .relationships(RelationshipType.withName("B"), Direction.OUTGOING);
return td.traverse(startNode);
```

Using `Evaluators`

An `Evaluator` takes the `Path` from the start node to the current node and decides whether the `Path` should be:

- Included in the result.
- Expanded for further evaluation.

Given `Path`, an evaluator can take one of four actions:

- `Evaluation.INCLUDE_AND_CONTINUE` — Include the current node in the result and continue the traversal.
- `Evaluation.INCLUDE_AND_PRUNE` — Include the current node in the result, but do not continue the traversal.
- `Evaluation.EXCLUDE_AND_CONTINUE` — Exclude the current node from the result, but continue the traversal.
- `Evaluation.EXCLUDE_AND_PRUNE` — Exclude the current node from the result and do not continue the traversal.

As more than one evaluator can be added, different combinations might lead to unexpected outcomes. For this reason, keep in mind that:

- A path is included if all evaluators include it.
- A path can only be expanded if none of the evaluators prune it.
- Evaluators are called for all paths the traverser encounters, even for the path that consists of the start node only.

Built-in Evaluators

The Traversal Framework provides several built-in evaluators:

Evaluator	Description
`Evaluators.all()`	Includes and continues on all nodes.
`Evaluators.atDepth(int)`	Includes only paths at the given depth, pruning everything else.
`Evaluators.toDepth(int)`	Includes paths up to the given depth, pruning everything deeper.
`Evaluators.fromDepth(int)`	Includes paths from the given depth, ignoring those before and never prunes anything.
`Evaluators.includingDepths(int, int)`	Includes only the paths of depth equal to and between the 2 given depths.
`Evaluators.lastRelationshipTypeIs(Evaluation, Evaluation, RelationshipType…)`	Allows the choice of which evaluation to take based on whether the last relationship matches one of the given ones.
`Evaluators.includeWhereLastRelationshipTypeIs(RelationshipType…)`	Only returns paths in which the final relationship matches the given ones.
`Evaluators.endNodeIs(Evaluation, Evaluation, Node…)`	Allows the choice of which evaluation to take based on whether the last node matches one of the given nodes.
`Evaluators.includeIfContainsAll(Node…)`	Returns a path if all given nodes are contained within it.
`Evaluators.includeIfAcceptedByAny(PathEvaluator)`	Returns a path if any of the given evaluators include the current path.
`Evaluators.endNodeIsAtDepth(int, Node…)`	Returns a path if one of the given nodes is at the given depth.

Here is an example of how to use a built-in evaluator:

```
TraversalDescription td;
try ( Transaction tx = graphDb.beginTx() ) {
     td = tx.traversalDescription()
            .evaluator(Evaluators.atDepth(2));
}

td.traverse( startNode );
```

Custom implementations of Evaluator

Now you can see an example of a custom implementation that includes only paths that end with a node of a certain label:

```
class LabelEvaluator implements Evaluator {

    private final Label label;

    private LabelEvaluator(Label label) {
        this.label = label;
    }

    @Override
    public Evaluation evaluate(Path path) {
        if (path.endNode().hasLabel(label)) {
            return Evaluation.INCLUDE_AND_CONTINUE;
        } else {
            return Evaluation.EXCLUDE_AND_CONTINUE;
        }
    }
}
```

The following example features a combined evaluator, which returns all paths of length 2 that also have an end node with the label A:

```
TraversalDescription td;
try ( Transaction tx = graphDb.beginTx() ) {
     td = tx.traversalDescription()
            .evaluator(Evaluators.atDepth( 2 ))
            .evaluator(new LabelEvaluator(Label.label("A")));
}

td.traverse( startNode );
```

Uniqueness options

Although the default is NODE_GLOBAL, it is possible to set the rules for how positions can be revisited during a traversal by adjusting the levels of Uniqueness. These are some of the available options:

- NONE — Any node in the graph may be revisited.
- NODE_GLOBAL — No node in the entire graph may be visited more than once. This could potentially consume a lot of memory since it requires keeping an in-memory data structure remembering all the visited nodes.
- RELATIONSHIP_GLOBAL — No relationship in the entire graph may be visited more than once. Just like NODE_GLOBAL, this could potentially use up a lot of memory. However, since graphs typically have a larger number of relationships than nodes, the memory overhead of this Uniqueness level could grow even quicker.
- NODE_PATH — A node may not occur previously in the path reaching up to it.
- RELATIONSHIP_PATH — A relationship may not occur previously in the path reaching up to it.
- NODE_RECENT — Similar to NODE_GLOBAL when it comes to using a global collection of visited nodes each position is checked against. However, this Uniqueness level has a cap on how much memory it may consume in the form of a collection that only contains the most recently visited nodes. The size of this collection can be specified by providing a number as the second argument to the

TraversalDescription.uniqueness()-method along with the Uniqueness level.

- RELATIONSHIP_RECENT — Works like NODE_RECENT, but with relationships instead of nodes.

Here is an example of a traversal using a built-in Uniqueness constraint:

```
TraversalDescription td;
try ( Transaction tx = graphDb.beginTx() ) {
     td = tx.traversalDescription();
            .uniqueness( Uniqueness.RELATIONSHIP_GLOBAL )
}

td.traverse( startNode );
```

BranchOrderingPolicy and BranchSelector

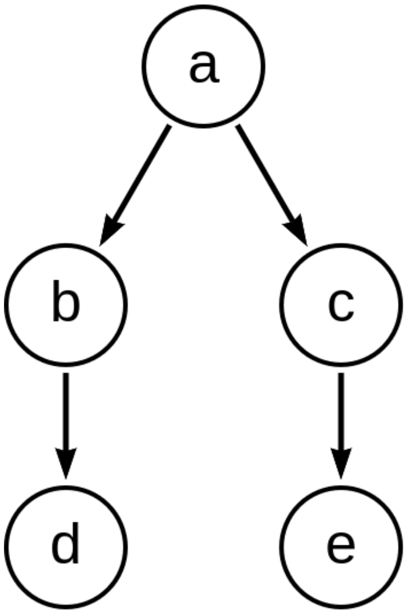
BranchOrderingPolicy is a factory for creating BranchSelectors to decide in what order branches are returned — that is, where a branch's position is represented as Path from the start node to the current node.

The Traversal Framework provides a few basic ordering implementations based on the depth-first and breadth-first algorithms. These are:

- BranchOrderingPolicies.PREORDER_DEPTH_FIRST — Traversing depth first, visiting each node before visiting its child nodes.
- BranchOrderingPolicies.POSTORDER_DEPTH_FIRST — Traversing depth first, visiting each node after visiting its child nodes.
- BranchOrderingPolicies.PREORDER_BREADTH_FIRST — Traversing breadth first, visiting each node before visiting its child nodes.
- BranchOrderingPolicies.POSTORDER_BREADTH_FIRST — Traversing breadth first, visiting each node after visiting its child nodes.

> Keep in mind that breadth-first traversals have a higher memory overhead than depth-first traversals.

The following example shows the result of BranchOrderingPolicy without any extra filter:



Ordering policy	Order of the nodes in traversal
`BranchOrderingPolicies.PREORDER_DEPTH_FIRST`	a, b, d, c, e
`BranchOrderingPolicies.POSTORDER_DEPTH_FIRST`	d, b, e, c, a
`BranchOrderingPolicies.PREORDER_BREADTH_FIRST`	a, b, c, d, e
`BranchOrderingPolicies.POSTORDER_BREADTH_FIRST`	d, e, b, c, a

Depth-first and breadth-first are common policies and can be accessed by the convenience methods `breadthFirst()` and `depthFirst()`. This is equivalent to setting the `BranchOrderingPolicies.PREORDER_BREADTH_FIRST` / `BranchOrderingPolicies.PREORDER_DEPTH_FIRST` policy.

Example of using `depthFirst()`:

```
TraversalDescription td;
try ( Transaction tx = graphDb.beginTx() ) {
     td = tx.traversalDescription()
            .depthFirst();
}

td.traverse( startNode );
```

Example of using `BranchOrderingPolicies.PREORDER_BREADTH_FIRST`:

```
TraversalDescription td;
try ( Transaction tx = graphDb.beginTx() ) {
     td = tx.traversalDescription()
            .order( BranchOrderingPolicies.PREORDER_BREADTH_FIRST );
}

td.traverse( startNode );
```

Since `BranchSelector` carries state and hence needs to be uniquely instantiated for each traversal, it should be supplied to the `TraversalDescription` through a `BranchOrderingPolicy` interface, which is a factory of `BranchSelector` instances.

Even though a user of the Traversal Framework rarely needs to implement their own `BranchSelector`/

`BranchOrderingPolicy`, it is relevant to know that these parameters let graph algorithm implementors provide their own traversal orders.

Check the Neo4j Graph Algorithms package to see `BestFirst` order `BranchSelector` / `BranchOrderingPolicy` that is used in `BestFirst` search algorithms such as A* and Dijkstra.

Using `PathExpander`

The Traversal Framework uses `PathExpander` to discover the relationships that should be followed from a particular path to further branches in the traversal.

There are multiple ways of specifying `PathExpander`, such as:

- The built-in `PathExpander` defines some commonly used `PathExpander`s.
- The `PathExpanderBuilder` allows the combination of definitions.
- It is possible to write a custom `PathExpander` by implementing the `PathExpander` interface.

Built-in `PathExpanders`

The following path expanders are found in the class `PathExpander` and can be used to set a more specific `PathExpander` for the traversal:

- `allTypesAndDirections()` — Expands all relationships in all directions (default).
- `forType(relationshipType)` — Expands only relationships of a specific type.
- `forDirection(direction)` — Expands only relationships in a specific direction.
- `forTypeAndDirection(relationshipType, direction)` — Expands only relationships of a given type and a given direction.
- `forTypesAndDirections(relationshipType, direction, relationshipType, direction, … )` — Expands only relationships of the given types and their specific direction.
- `forConstantDirectionWithTypes(relationshipType, …)` — Expands only relationships of the given types, if they continue in the direction of the first relationship.

Here is an example of how to set a custom relationship expander that only expands outgoing relationships with the type `A`:

```
TraversalDescription td = transaction.traversalDescription()
    .expand(PathExpanders.forTypeAndDirection( RelationshipType.withName( "A" ), Direction.OUTGOING ));
td.traverse( startNode );
```

`PathExpanderBuilder`

The `PathExpanderBuilder` allows the combination of different `PathExpander` definitions. It provides a more fine-grained level of customization without the need to write `PathExpander` from scratch. It also contains a set of static methods allowing the creation of `PathExpander` with the following methods:

- `empty()` — Expands no relationships.

- emptyOrderedByType() — Expands no relationships and guarantees the order of how types will be expanded when any are added.
- allTypesAndDirections() — Expands all relationships in any direction.
- allTypes(Direction) — Expands all relationships in the given direction.

That PathExpander can then be further defined by the following methods:

- add(relationshipType) — Expands relationships of the given type.
- add(relationshipType, direction) — Expands relationships of the given type and direction.
- remove(relationshipType) — Removes the expansion of relationships of the given type.
- addNodeFilter(filter) — Adds a filter based on nodes.
- addRelationshipFilter(filter) — Adds a filter based on relationships.

This is how it may look:

```
TraversalDescription td = transaction.traversalDescription()
    .expand(PathExpanderBuilder.empty()
                               .add(RelationshipType.withName("E1"))
                               .build());
td.traverse( startNode );
```

The following is an example of a custom PathExpander which tracks the weight of the path in its BranchState and only includes paths if the total weight is smaller than the given maximum weight:

```
class MaxWeightPathExpander implements PathExpander<Double>
{

    private final double maxWeight;

    public MaxWeightPathExpander( double maxWeight ) {
        this.maxWeight = maxWeight;
    }

    @Override
    public Iterable<Relationship> expand( Path path, BranchState<Double> branchState )
    {
        if (path.lastRelationship() != null) {
            branchState.setState( branchState.getState() + (double) path.lastRelationship().getProperty(
"weight" ) );
        }

        Iterable<Relationship> relationships = path.endNode().getRelationships( Direction.OUTGOING );
        ArrayList<Relationship> filtered = new ArrayList<>();
        for ( Relationship relationship : relationships ) {
            if ( branchState.getState() + (double) relationship.getProperty( "weight" ) <= maxWeight ) {
                filtered.add(relationship);
            }
        }
        return filtered;
    }

    @Override
    public PathExpander reverse()
    {
        throw new RuntimeException( "Not needed for the MonoDirectional Traversal Framework" );
    }
}
```

Here is an example of how to use the custom PathExpander and set the initial state:

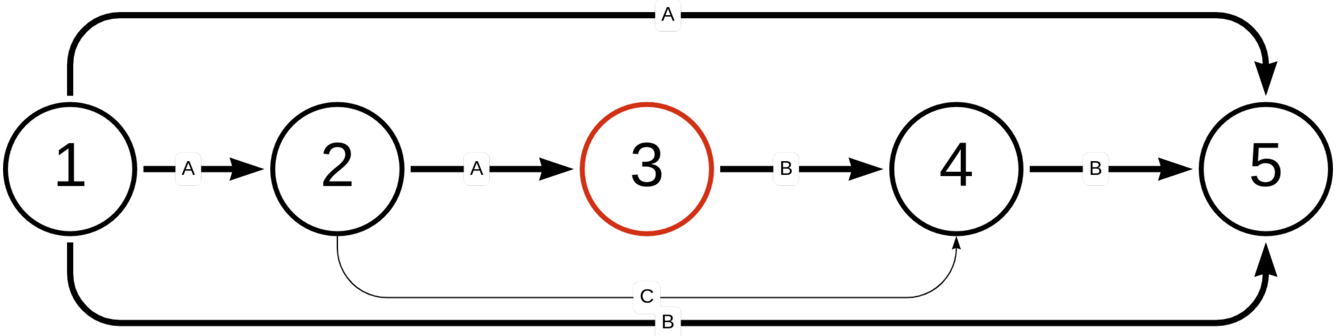
```
TraversalDescription td = transaction.traversalDescription()
        .expand( new MaxWeightPathExpander(5.0), InitialBranchState.DOUBLE_ZERO );
td.traverse( startNode );
```

Bidirectional Traversal Framework

A bidirectional traversal consists of two traversals starting from different nodes and returning the paths to where both traversals collide. The Bidirectional Traversal Framework allows the description of such a traversal with the `BidirectionalTraversalDescription`.

The following is a visual representation of a bidirectional traversal from start node `1` and end node `5`, where all bold relationships are in the result paths given the following restrictions:

- The start side traverser only traverses relationships of type `A`.
- The end side traverser only traverses relationships of type `B`.



Note that paths where only one of the traversers reaches the opposite start/end node are also included, such as `(1)-[:A]→(5)` and `(1)-[:B]→(5)`. In the accepted path `(1)-[:A]→(2)-[:A]→(3)-[:B]→(4)-[:B]→(5)`, the traversers collide in the middle at node `3`.

> The `Uniqueness` is shared between both traversals, which means that there will be no results if the `Uniqueness` is set to `Uniqueness.NODE_GLOBAL` (which is the default) as both traversals need to reach the same node to cause a collision. Therefore, when using `BidirectionalTraversalDescription`, always set the `Uniqueness` manually. For more information on the available options, see `Uniqueness` options.

Defining the bidirectional traverser

When creating `BidirectionalTraversalDescription`, it is possible to choose the `TraversalDescription` that each side takes. This can be done by setting it individually for each side (`startSide`/`endSide`) or setting one for both (`mirroredSides`).

Start and end side traversal

The following is an example of `BidirectionalTraversalDescription` with a separate traverser for the `startSide()` and `endSide()`:

```
BidirectionalTraversalDescription td = transaction
        .bidirectionalTraversalDescription()
        .startSide(transaction
                .traversalDescription()
                .expand(PathExpanders.forTypeAndDirection(RelationshipType.withName("A"), Direction
.OUTGOING))
                .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL))
        .endSide(transaction
                .traversalDescription()
                .expand(PathExpanders.forTypeAndDirection(RelationshipType.withName("B"), Direction
.INCOMING))
                .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL));
td.traverse(startNode, endNode);
```

One side is traversing forward from the start node and expanding all outgoing relationships of Type `A`. The other side is traversing backward from the end node and expanding all incoming relationships of Type `B`.

The final paths go from the start node to the end node, where all relationships have an outgoing direction. Possible paths are:

- All relationships are of Type `A`.
- All relationships are of Type `B`.
- The relationships from the start node are Type `A` and, after the collision, they are all of type `B`.

Mirrored traversal

The `mirroredSides(TraversalDescription)` method sets both the start side and end side of this bidirectional traversal. However, the end side is the reverse of the start. For the `Direction` of `Relationships`, this means that `OUTGOING` becomes `INCOMING` and vice versa. The `PathExpander` interface also comes with a `reverse()` function, which should be overridden if used in a `mirroredSides` implementation.

```
BidirectionalTraversalDescription td = transaction
        .bidirectionalTraversalDescription()
        .mirroredSides(transaction
                               .traversalDescription()
                               .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL));
td.traverse(startNode, endNode);
```

Side selector

In a bidirectional traversal, the traverser selects which side (start or end) to move on each step. It is possible to change how this works by implementing the `SideSelectorPolicy` interface, as it has a function for determining from which side to traverse the next step.

- `Direction.OUTGOING` — The traverser coming from the start node.
- `Direction.INCOMING` — The traverser coming from the end node.

The built-in policies include:

- `SideSelectorPolicies.LEVEL` — Stops traversal if the combined depth is larger than the given maximum depth.
- `SideSelectorPolicies.LEVEL_STOP_DESCENT_ON_RESULT` — Stops as soon as a result is found.

- `SideSelectorPolicies.ALTERNATING` — Alternates which branch continues the traversal.

The `SideSelectorPolicy` optionally takes `maxDepth`, which represents the combined depth that both sides must adhere to.

The following is an example of how to use the built-in `SideSelectorPolicy` with a max combined depth of 5:

```
BidirectionalTraversalDescription td = transaction
        .bidirectionalTraversalDescription()
        .mirroredSides(transaction
                              .traversalDescription()
                              .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL))
                              .sideSelector(SideSelectorPolicies.LEVEL, 5);
td.traverse(startNode, endNode);
```

Collision policies

`BranchCollisionPolicy` defines when a collision is detected and accepted in a bidirectional traversal. Given an evaluator and a path predicate, `BranchCollisionPolicy` creates `BranchCollisionDetector`, which detects collisions between two traversers and adds the resulting path to the result if it satisfies the conditions given by the collision evaluator and Uniqueness constraints.

These are the built-in `BranchCollisionPolicies`:

- `STANDARD` — Returns all paths with a collision.
- `SHORTEST_PATH` — Returns all paths with the smallest traversal depth.

The following is an example of how to use the built-in `BranchCollisionPolicy`:

```
BidirectionalTraversalDescription td = transaction
        .bidirectionalTraversalDescription()
        .mirroredSides(transaction
                              .traversalDescription()
                              .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL))
        .collisionPolicy(BranchCollisionPolicies.SHORTEST_PATH);
td.traverse(startNode, endNode);
```

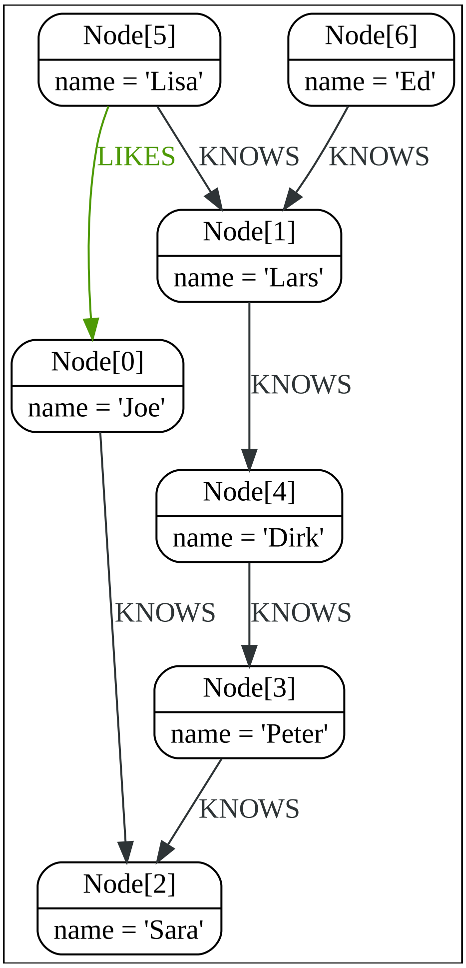
Collision evaluator

The convenience method `collisionEvaluator()` adds `PathEvaluator`, which validates paths in a valid collision. Using the method multiple times results in the combination of the evaluators.

```
BidirectionalTraversalDescription td = transaction
    .bidirectionalTraversalDescription()
    .mirroredSides(transaction
       .traversalDescription()
       .uniqueness(Uniqueness.RELATIONSHIP_GLOBAL))
    .collisionEvaluator(Evaluators.atDepth(3));
td.traverse(startNode, endNode);
```

Traversal Framework examples

The following are some examples of how you can use the Traversal Framework. The source code for the examples can be found in `TraversalExample.java`.



This graph illustrates a small group of friends with the definition of `RelationshipType`:

```
private enum Rels implements RelationshipType
{
    LIKES, KNOWS
}
```

Traversing a graph examples

The graph can be traversed with, for example, the following traverser starting at the node with the `name = 'Joe'`:

```
for ( Path position : db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .relationships( Rels.LIKES, Direction.INCOMING )
        .evaluator( Evaluators.toDepth( 5 ) )
        .traverse( node ) )
{
    output += position + "\n";
}
```

The traversal will thus output:

```
(0)
(0)<-[LIKES,1]-(5)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)<-[KNOWS,5]-(6)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)-[KNOWS,4]->(4)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)-[KNOWS,4]->(4)-[KNOWS,3]->(3)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)-[KNOWS,4]->(4)-[KNOWS,3]->(3)-[KNOWS,2]->(2)
```

Since `TraversalDescription` is immutable, it is useful to create template descriptions that include common settings shared by different traversals. For example, start with this traverser:

```
friendsTraversal = db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .uniqueness( Uniqueness.RELATIONSHIP_GLOBAL );
```

It yields the following output (starting from the node with the `name = 'Joe'`):

```
(0)
(0)-[KNOWS,0]->(2)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)<-[KNOWS,6]-(5)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)<-[KNOWS,5]-(6)
```

Then, create a new traverser from it, restricting depth to three:

```
for ( Path path : friendsTraversal
        .evaluator( Evaluators.toDepth( 3 ) )
        .traverse( node ) )
{
    output += path + "\n";
}
```

This should be the output:

```
(0)
(0)-[KNOWS,0]->(2)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)
```

In case you want to traverse from depth 2 to 4:

```
for ( Path path : friendsTraversal
        .evaluator( Evaluators.fromDepth( 2 ) )
        .evaluator( Evaluators.toDepth( 4 ) )
        .traverse( node ) )
{
    output += path + "\n";
}
```

This gives the following output:

```
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)
```

For other useful evaluators, see Using Evaluators.

The `Traverser` also has a `nodes()` method, which returns an `Iterable` of all the nodes in the paths:

```
for ( Node currentNode : friendsTraversal
        .traverse( node )
        .nodes() )
{
    output += currentNode.getProperty( "name" ) + "\n";
}
```

This gives the following output:

```
Joe
Sara
Peter
Dirk
Lars
Lisa
Ed
```

You can do this with relationships as well. Here is an example:

```
for ( Relationship relationship : friendsTraversal
        .traverse( node )
        .relationships() )
{
    output += relationship.getType().name() + "\n";
}
```

```
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
```

Implementing a user-defined procedure

This example shows how to implement a user-defined procedure using the Traversal Framework. The transaction and logger are made available through the Procedure Framework:

```java
@Context
public Transaction tx;

@Context
public Log log;

@Procedure(value = "traverse.findPeople")
@Description("Finds all the known people to the given Person")
public Stream<PathResult> findFriends(@Name("person") Node person) {

    final Traverser traverse = tx.traversalDescription()
            .breadthFirst()
            .relationships(RelationshipType.withName("KNOWS"), Direction.OUTGOING)
            .evaluator(Evaluators.toDepth(5))
            .evaluator(new PathLogger())
            .traverse(person);

    return stream(traverse.iterator()).map(PathResult::new);
}

private final class PathLogger implements Evaluator {

    @Override
    public Evaluation evaluate(Path path) {
        log.info(path.toString());
        return Evaluation.INCLUDE_AND_CONTINUE;
    }
}
```

This allows the Traversal Framework to be used side by side with Cypher:

```cypher
MATCH (p:Person { name: 'Joe' })
CALL traverse.findPeople(p) YIELD path RETURN [friend IN nodes(path) | friend.name] AS friends
```

Transaction management

This topic describes transactional management and behavior.

To fully maintain data integrity and ensure good transactional behavior, Neo4j DBMS supports the **ACID** properties:

- Atomicity — If a part of a transaction fails, the database state is left unchanged.
- Consistency — Every transaction leaves the database in a consistent state.
- Isolation — During a transaction, modified data cannot be accessed by other operations.
- Durability — The DBMS can always recover the results of a committed transaction.

Specifically:

- All database operations that access the graph, indexes, or schema must be performed in a transaction.
- The default isolation level is *read-committed isolation level*.
- Data retrieved by traversals is not protected from modification by other transactions.
- Non-repeatable reads may occur (i.e., only write locks are acquired and held until the end of the transaction).
- One can manually acquire write locks on nodes and relationships to achieve a higher level of isolation — *serialization isolation level*.
- Locks are acquired at the Node and Relationship levels.
- Deadlock detection is built into the core transaction management.

Interaction cycle

There are database operations that must be performed in a transaction to ensure the ACID properties. Specifically, operations that access the graph, indexes, or schema are such operations. Transactions are single-threaded, confined, and independent. Multiple transactions can be started in a single thread and they are independent of each other.

The interaction cycle of working with transactions follows the steps:

1. Begin a transaction.
2. Perform database operations.
3. Commit or roll back the transaction.

> It is crucial to finish each transaction. The locks or memory acquired by a transaction are only released upon completion.

The idiomatic use of transactions in Neo4j is to use a `try-with-resources` statement and declare `transaction` as one of the resources. Then start the transaction and try to perform graph operations. The last operation in the `try` block should commit or roll back the transaction, depending on the business logic. In this scenario, `try-with-resources` is used as a guard against unexpected exceptions and as an additional safety mechanism to ensure that the transaction gets closed no matter what happens inside the statement block. All non-committed transactions will be rolled back as part of resource cleanup at the end of the statement. No resource cleanup is required for a transaction that is explicitly committed or rolled back, and the transaction closure is an empty operation.

> All modifications performed in a transaction are kept in memory. This means that very large updates must be split into several transactions to avoid running out of memory.

Isolation levels

Transactions in Neo4j use a *read-committed isolation level*, which means they see data as soon as it has been committed but cannot see data in other transactions that have not yet been committed. This type of isolation is weaker than serialization but offers significant performance advantages while being sufficient for the overwhelming majority of cases.

In addition, the Neo4j Java API enables explicit locking of nodes and relationships. Using locks allows simulating the effects of higher levels of isolation by obtaining and releasing locks explicitly. For example, if a write lock is taken on a common node or relationship, then all transactions are serialized on that lock — giving the effect of a *serialization isolation level*.

Lost updates in Cypher

In Cypher it is possible to acquire write locks to simulate improved isolation in some cases. Consider the case where multiple concurrent Cypher queries increment the value of a property. Due to the limitations of the *read-committed isolation level*, the increments might not result in a deterministic final value. If there is a direct dependency, Cypher automatically acquires a write lock before reading. A direct dependency is when the right-hand side of `SET` has a dependent property read in the expression or the value of a key-value pair in a literal map.

For example, if you run the following query by one hundred concurrent clients, it is very likely not to increment the property `n.prop` to 100, unless a write lock is acquired before reading the property value. This is because all queries read the value of `n.prop` within their own transaction, and cannot see the incremented value from any other transaction that has not yet been committed. In the worst case scenario, the final value would be as low as 1 if all threads perform the read before any has committed their transaction.

Example 1. Cypher can acquire a write lock

The following example requires a write lock, and Cypher automatically acquires one:

```
MATCH (n:Example {id: 42})
SET n.prop = n.prop + 1
```

Example 2. Cypher can acquire a write lock

This example also requires a write lock, and Cypher automatically acquires one:

```
MATCH (n)
SET n += {prop: n.prop + 1}
```

Due to the complexity of determining such a dependency in the general case, Cypher does not cover any of the following example cases:

Example 3. Complex Cypher

Variable depending on results from reading the property in an earlier statement:

```
MATCH (n)
WITH n.prop AS p
// ... operations depending on p, producing k
SET n.prop = k + 1
```

Example 4. Complex Cypher

Circular dependency between properties read and written in the same query:

```
MATCH (n)
SET n += {propA: n.propB + 1, propB: n.propA + 1}
```

To ensure deterministic behavior also in the more complex cases, it is necessary to explicitly acquire a write lock on the node in question. In Cypher there is no explicit support for this, but it is possible to work around this limitation by writing to a temporary property.

Example 5. Explicitly acquire a write lock

This example acquires a write lock for the node by writing to a dummy property before reading the requested value:

```
MATCH (n:Example {id: 42})
SET n._LOCK_ = true
WITH n.prop AS p
// ... operations depending on p, producing k
SET n.prop = k + 1
REMOVE n._LOCK_
```

The existence of the `SET n._LOCK_` statement before the read of the `n.prop` read ensures the lock is acquired before the read action, and no updates are lost due to enforced serialization of all concurrent queries on that specific node.

Default locking behavior

- When adding, changing, or removing a property on a node or relationship a write lock is taken on the specific node or relationship.
- When creating or deleting a node a write lock is taken for the specific node.
- When creating or deleting a relationship a write lock is taken on the specific relationship and both its nodes.

The locks are added to the transaction and released when the transaction finishes.

Deadlocks

Since locks are used, deadlocks can happen. Neo4j, however, detects any deadlock (caused by acquiring a lock) before they happen and throws an exception. The transaction is marked for rollback before the exception is thrown. All locks acquired by the transaction are still held but will be released when the transaction finishes. Once the locks are released, other transactions that were waiting for locks held by the transaction causing the deadlock can proceed. You can then retry the work performed by the transaction causing the deadlock if needed.

Experiencing frequent deadlocks is an indication of concurrent write requests happening in such a way that it is not possible to execute them while at the same time living up to the intended isolation and consistency. The solution is to make sure concurrent updates happen reasonably. For example, given two specific nodes (A and B), adding or deleting relationships to both these nodes in random order for each transaction results in deadlocks when two or more transactions do that concurrently. One option is to make sure that updates always happen in the same order (first A then B). Another option is to make sure that each thread/transaction does not have any conflicting writes to a node or relationship as some other concurrent transaction. This can, for example, be achieved by letting a single thread do all updates of a specific type.

> Deadlocks caused by the use of other synchronization than the locks managed by Neo4j can still happen. Since all operations in the Neo4j API are thread-safe unless specified otherwise, there is no need for external synchronization. Other code that requires synchronization should be synchronized in such a way that it never performs any Neo4j operation in the synchronized block.

Deadlock handling an example

The following is an example of how deadlocks can be handled in procedures, server extensions, or when using Neo4j embedded.

> The full source code used for the code snippet can be found in DeadlockDocTest.java.

When dealing with deadlocks in code, there are several issues you may want to address:

- Only do a limited amount of retries, and fail if a threshold is reached.
- Pause between each attempt to allow the other transaction to finish before trying again.

- A retry loop can be useful not only for deadlocks but for other types of transient errors as well.

Below is an example that shows how this can be implemented.

Example 6. Handling deadlocks using a retry loop

This example shows how to use a retry loop for handling deadlocks:

```
Throwable txEx = null;
int RETRIES = 5;
int BACKOFF = 3000;
for ( int i = 0; i < RETRIES; i++ )
{
    try ( Transaction tx = databaseService.beginTx() )
    {
        Object result = doStuff(tx);
        tx.commit();
        return result;
    }
    catch ( Throwable ex )
    {
        txEx = ex;

        // Add whatever exceptions to retry on here
        if ( !(ex instanceof DeadlockDetectedException) )
        {
            break;
        }
    }

    // Wait so that we don't immediately get into the same deadlock
    if ( i < RETRIES - 1 )
    {
        try
        {
            Thread.sleep( BACKOFF );
        }
        catch ( InterruptedException e )
        {
            throw new TransactionFailureException( "Interrupted", e );
        }
    }
}

if ( txEx instanceof TransactionFailureException )
{
    throw ((TransactionFailureException) txEx);
}
else if ( txEx instanceof Error )
{
    throw ((Error) txEx);
}
else
{
    throw ((RuntimeException) txEx);
}
```

Delete semantics

When deleting a node or a relationship all properties for that entity will be automatically removed but the relationships of a node will not be removed. Neo4j enforces a constraint (upon commit) that all relationships must have a valid start node and end node. In effect, this means that trying to delete a node that still has relationships attached to it will throw an exception upon commit. It is, however, possible to choose in which order to delete the node and the attached relationships as long as no relationships exist when the transaction is committed.

The delete semantics can be summarized as follows:

- All properties of a node or relationship will be removed when it is deleted.
- A deleted node cannot have any attached relationships when the transaction commits.
- It is possible to acquire a reference to a deleted relationship or node that has not yet been committed.
- Any write operation on a node or relationship after it has been deleted (but not yet committed) will throw an exception.
- Trying to acquire a new or work with an old reference to a deleted node or relationship after commit, will throw an exception.

Creating unique nodes

In many use cases, a certain level of uniqueness is desired among entities. For example, only one user with a certain email address may exist in a system. If multiple concurrent threads naively try to create the user, duplicates will be created.

The following are the main strategies for ensuring uniqueness, and they all work across cluster and single-instance deployments.

Single thread

By using a single thread, no two threads even try to create a particular entity simultaneously. In a cluster, an external single-threaded client can perform the operations.

Get or create

Defining a uniqueness constraint and using the Cypher `MERGE` clause is the most efficient way to get or create a unique node. See Unique nodes for more information.

Transaction events

A `neo4j.org.graphdb.event.TransactionEventListener` can be registered to receive Neo4j database transaction events. Once it has been registered at a `org.neo4j.dbms.api.DatabaseManagementService` instance, it receives transaction events for the database with which it was registered. Listeners get notified about transactions that have performed any write operation, and that will be committed. If `Transaction#commit()` has not been called, or the transaction was rolled back with `Transaction#rollback()`, it will be rolled back and no events are sent to the listener.

Before a transaction is committed, the listeners' `beforeCommit` method is called with the entire diff of modifications made in the transaction. At this point the transaction is still running, so changes can still be made. The method may also throw an exception, which prevents the transaction from being committed. If the transaction is rolled back, a call to the listener's `afterRollback` method will follow.

	The order in which listeners are executed is undefined — there is no guarantee that changes made by one listener will be seen by other listeners.

If `beforeCommit` is successfully executed in all registered listeners, the transaction is committed and the `afterCommit` method is called with the same transaction data. This call also includes the object returned from `beforeCommit`.

In `afterCommit`, the transaction is closed and access to anything outside `org.neo4j.graphdb.event.TransactionData` requires a new transaction to be opened. A `neo4j.org.graphdb.event.TransactionEventListener` gets notified about transactions that have any changes accessible via `org.neo4j.graphdb.event.TransactionData`. Some indexing and schema changes will not trigger these events.

The following example shows how to register a listener for a specific database and perform basic operations on top of the transaction change set.

	The full source code used for the code snippet can be found in TransactionEventListenerExample.java.

Example 7. TransactionEventListener

Register a transaction event listener and inspect the change set:

```
public static void main( String[] args ) throws IOException
{
    FileUtils.deleteDirectory( HOME_DIRECTORY );
    var managementService = new DatabaseManagementServiceBuilder( HOME_DIRECTORY ).build();
    var database = managementService.database( DEFAULT_DATABASE_NAME );

    var countingListener = new CountingTransactionEventListener();
    managementService.registerTransactionEventListener( DEFAULT_DATABASE_NAME, countingListener );

    var connectionType = RelationshipType.withName( "CONNECTS" );
    try ( var transaction = database.beginTx() )
    {
        var startNode = transaction.createNode();
        var endNode = transaction.createNode();
        startNode.createRelationshipTo( endNode, connectionType );
        transaction.commit();
    }
}

private static class CountingTransactionEventListener implements TransactionEventListener
<CreatedEntitiesCounter>
{
    @Override
    public CreatedEntitiesCounter beforeCommit( TransactionData data, Transaction transaction,
GraphDatabaseService databaseService ) throws Exception
    {
        return new CreatedEntitiesCounter( size( data.createdNodes() ), size( data
.createdRelationships() ) );
    }

    @Override
    public void afterCommit( TransactionData data, CreatedEntitiesCounter entitiesCounter,
GraphDatabaseService databaseService )
    {
        System.out.println( "Number of created nodes: " + entitiesCounter.getCreatedNodes() );
        System.out.println( "Number of created relationships: " + entitiesCounter
.getCreatedRelationships() );
    }

    @Override
    public void afterRollback( TransactionData data, CreatedEntitiesCounter state,
GraphDatabaseService databaseService )
    {
    }
}

private static class CreatedEntitiesCounter
{
    private final long createdNodes;
    private final long createdRelationships;

    public CreatedEntitiesCounter( long createdNodes, long createdRelationships )
    {
        this.createdNodes = createdNodes;
        this.createdRelationships = createdRelationships;
    }

    public long getCreatedNodes()
    {
        return createdNodes;
    }

    public long getCreatedRelationships()
    {
        return createdRelationships;
    }
}
```

JMX metrics

This topic describes how to access JMX for Neo4j DBMS to monitor metrics.

Neo4j provides different levels of monitoring facilities to supply a continuous overview of the system's health. For a description of the monitoring options, see Neo4j Operations Manual → Monitoring. Many of the metrics are exposed through JMX.

> The available JMX MBeans and their names have been updated in Neo4j 4.0. Beans that duplicate metrics or monitoring options, described in Neo4j Operations Manual → Monitoring, have been removed.

Adjusting remote JMX access to Neo4j

By default, the Neo4j Enterprise edition does not allow remote JMX connections. To enable this feature, you need to enable JMX Remote Management and also configure JMX for secure remote access, in the conf/neo4j.conf file.

Enable JMX Remote Management

Add the following lines to the *conf/neo4j.conf* file to enable JMX Remote Management. If you run into issues with automatic hostname discovery, you can uncomment the following configuration line:

`server.jvm.additional=-Djava.rmi.server.hostname=$THE_NEO4J_SERVER_HOSTNAME`

```
server.jvm.additional=-Dcom.sun.management.jmxremote.port=3637
server.jvm.additional=-Dcom.sun.management.jmxremote.authenticate=true
server.jvm.additional=-Dcom.sun.management.jmxremote.ssl=false

# Some systems cannot discover the hostname automatically, and need this line configured:
# server.jvm.additional=-Djava.rmi.server.hostname=$THE_NEO4J_SERVER_HOSTNAME
```

> Although SSL for JMX Remote Management is disabled throughout this document, to configure it based on your requirements, you can follow the instructions in the Java SE 11 Monitoring and Management Guide.

Configure password authentication

Password authentication is enabled by default in JMX Remote Management. You can find information about setting up authentication with LDAP and file-based approach in the following sections.

Refer to the Java SE 11 Monitoring and Management Guide for more options, including configuration steps for SSL client authentication.

LDAP authentication

You can configure your JAAS login configuration based on your infrastructure and save it in the *conf/jmx.ldap* configuration file.

```
Neo4jJMXConfig {
    com.sun.security.auth.module.LdapLoginModule REQUIRED
    userProvider="ldap://127.0.0.1:10389/ou=users,dc=example,dc=net"
    authIdentity="uid={USERNAME},ou=users,dc=example,dc=net"
    userFilter="(&(samaccountname={USERNAME})(objectClass=inetOrgPerson))"
    useSSL=false
    debug=false
    authzIdentity=monitorRole;
};
```

`userProvider`
: Defines which LDAP server to connect and the node to perform the search against user entries.

`authIdentity`
: Defines the distinguished name of the user to authenticate to the LDAP server. Note that the token `{USERNAME}` is replaced with the provided user name during authentication.

`userFilter`
: Defines the search filter to be used while locating the user. Note that the token `{USERNAME}` is replaced with the provided user name during the search.

`useSSL`
: Defines whether to enable SSL for the underlying LDAP connection.

`debug`
: Defines whether to output debug info about the authentication session.

`authzIdentity`
: Specifies which access role an authenticated user will be granted.

> The provided configuration is an example and needs to be updated based on your infrastructure.

After finishing your JAAS configuration, configure JMX to use it by adding the following configuration items into *conf/neo4j.conf* file:

```
server.jvm.additional=-Dcom.sun.management.jmxremote.login.config=Neo4jJMXConfig
server.jvm.additional=-Djava.security.auth.login.config=/absolute/path/to/conf/jmx.ldap
```

With this setup, you can connect to JMX monitoring of the Neo4j server using `<IP-OF-SERVER>:3637`, with a valid username and password defined in your LDAP directory.

File-based authentication

> The file-based password authentication stores the password in clear text and is intended only for development use.

You can set your password for JMX remote access and save it in the *conf/jmx.password* configuration file. Note that on Unix-based systems, the *jmx.password* file needs to be owned by the user that runs the server, and has permissions set to `0600`.

```
monitorRole password_to_be_changed
```

Next, configure the access level and save it in *conf/jmx.access* configuration file.

```
monitorRole readonly
```

Finally, configure JMX to use the completed password and access files by adding the following configuration items into *conf/neo4j.conf* file:

```
server.jvm.additional=-Dcom.sun.management.jmxremote.password.file=/absolute/path/to/conf/jmx.password
server.jvm.additional=-Dcom.sun.management.jmxremote.access.file=/absolute/path/to/conf/jmx.access
```

With this setup, you can connect to JMX monitoring of the Neo4j server using `<IP-OF-SERVER>:3637`, with the username `monitor`, and the password `password_to_be_changed`.

Connecting to a Neo4j instance using JMX and JConsole

First, start your Neo4j instance, for example using:

```
$NEO4j_HOME/bin/neo4j start
```

Now, start JConsole with:

```
$JAVA_HOME/bin/jconsole
```

Connect to the process running your Neo4j database instance:

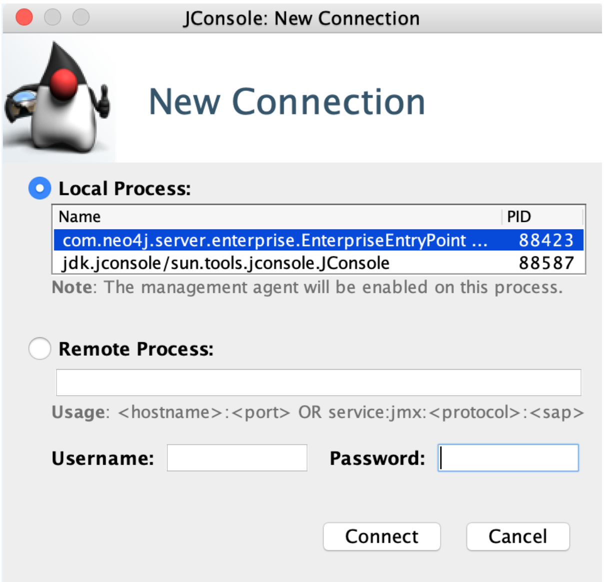


Figure 2. Connecting JConsole to the Neo4j Java process

Besides the MBeans, exposed by the JVM, you also see be default `neo4j.metrics` section in the MBeans tab. Under that, you have access to all the monitoring information exposed by Neo4j.

For opening JMX to remote monitoring access, please see Adjusting remote JMX access to Neo4j and the

[JMX documention](#).

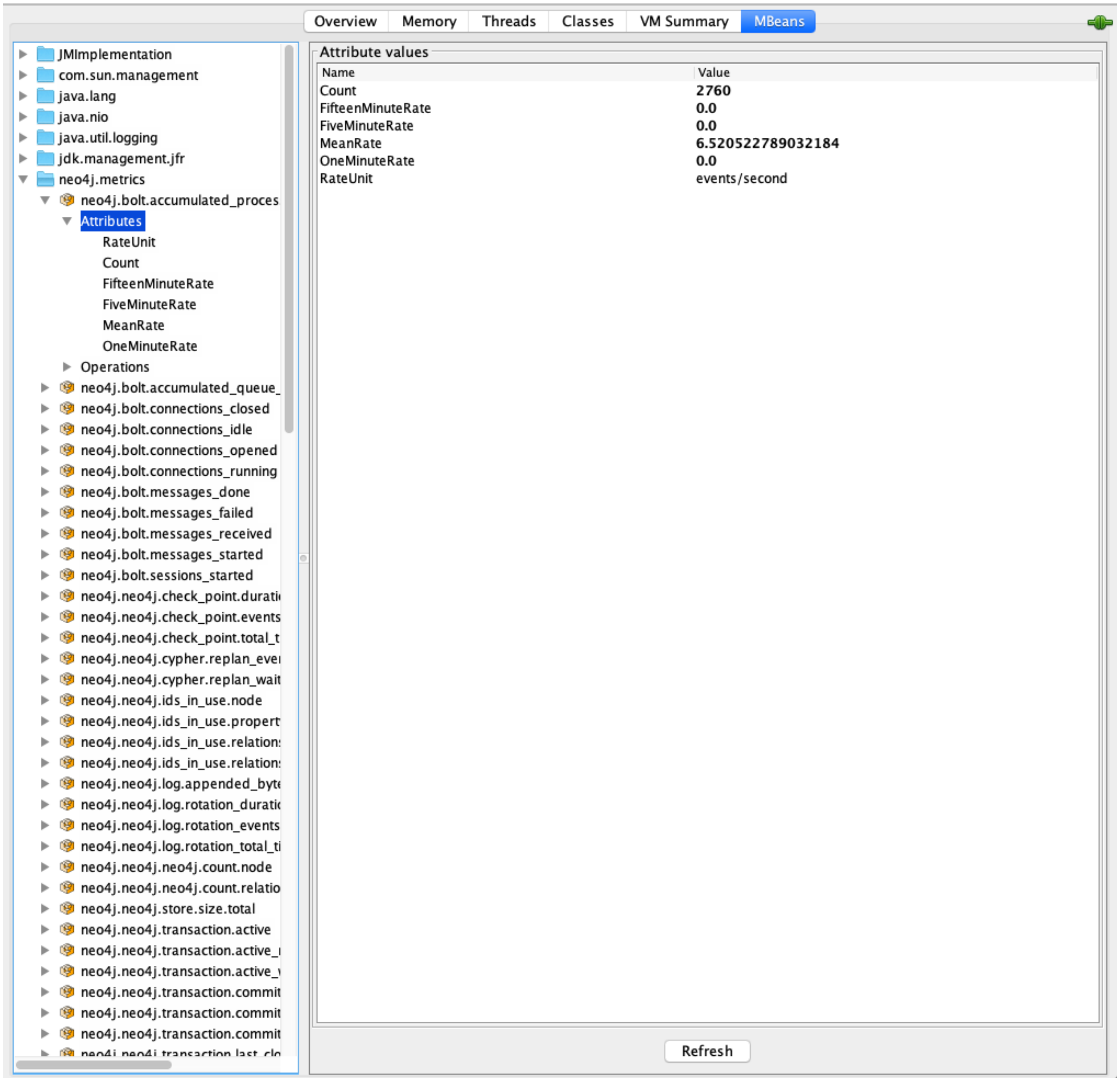


Figure 3. Neo4j MBeans view

License