2.2. Hello world

This section describes how to create and access nodes and relationships.

For information on project setup, see Section 2.1, “Including Neo4j in your project” (../include-neo4j/).

A Neo4j graph consists of:

- nodes
- relationships that connects 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.

The source code of this example is found here: EmbeddedNeo4j.java (https://github.com/neo4j/neo4j-documentation/blob/4.0/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4j.java)

2.2.1. 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 will make sure the database shuts down when the JVM exits. Next step is to interact with the database.

§ 2.2.2. Wrapping operations in a transaction

All operations have to be performed in a transaction. This is a deliberate design decision, since we believe transaction demarcation to be 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 Chapter 3, *Transaction management* (../../transaction-management/) and Java API for Transaction (https://neo4j.com/docs/java-reference/4.0/javadocs/org/neo4j/graphdb/Transaction.html).

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

2.2.3. Creating a small graph

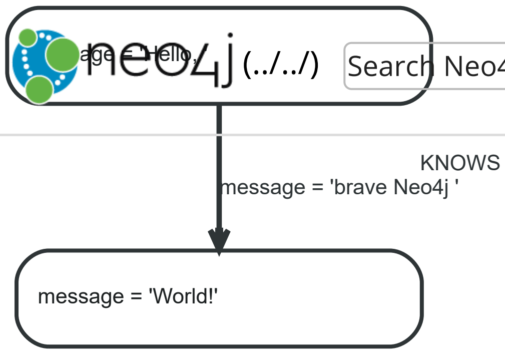
You can now create a few nodes. The API is very intuitive as you can see in the Neo4j Javadocs (https://neo4j.com/docs/java-reference/4.0/javadocs/) (they are included in the distribution as well). This is how to create a small graph consisting of two nodes, connected with one relationship and some properties:

```
            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:

Figure 2.1. Hello World Graph



2.2.4. 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 will output:

Hello, brave Neo4j World!

2.2.5. Removing the data

In this case, the data will be 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 which still has relationships when the transaction commits will fail. This is to make sure relationships always have a start node and an end node.

2.2.6. Shutting down the database server

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

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