Managing Data on the World Wide-Web

Neo4j – APIs and Transactions
Overview

1. REST API
2. Neo4j in Java
3. Transactions
There are two ways of using Neo4j from the JVM:

1. The standalone Neo4j Server can be installed on any machine and then accessed via its HTTP API by any REST library (e.g., Restlet)
   - Simple to learn and use

2. Embedding Neo4j in Java code by using its Java API
   - Object-oriented approach to the graph database
   - Better for testing
   - No-network setups
   - Optimized
REST API – Create Node

**Request**

```http
POST http://localhost:7474/db/data/node HTTP/1.1
Accept: application/json; charset=UTF-8
Content-Type: application/json; charset=UTF-8

{
  "foo": "bar"
}
```

**Response**

```http
HTTP/1.1 201 Created
Content-Length: 1237
Content-Type: application/json; charset=UTF-8
Location: http://localhost:7474/db/data/node/107

{
  "extensions": {},
  "labels": "http://localhost:7474/db/data/node/107/labels",
  "outgoing_relationships":
    "http://localhost:7474/db/data/node/107/relationships/out",
  ...
}
```
REST API – Get Node

GET http://localhost:7474/db/data/node/182 HTTP/1.1
Accept: application/json; charset=UTF-8

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: ...

{   "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/182/labels",
    "outgoing_relationships" :
        "http://localhost:7474/db/data/node/128/relationships/out",
    ...
}

What status code would we get if the requested node doesn’t exist?
# REST API – Create Relationship

**Request**

```
POST http://localhost:7474/db/data/node/1/relationships HTTP/1.1
Accept: application/json; charset=UTF-8
Content-Type: application/json; charset=UTF-8

{
    "to": "http://localhost:7474/db/data/node/0",
    "type": "LOVES"
}
```

**Response**

```
HTTP/1.1 201 Created
Content-Length: 1237
Content-Type: application/json; charset=UTF-8
Location: http://localhost:7474/db/db/data/relationship/1

{
    "start": "http://localhost:7474/db/data/node/1",
    "end": "http://localhost:7474/db/data/node/0",
    ...
}
```
REST API – Get Relationship

GET http://localhost:7474/db/data/relationship/46 HTTP/1.1
Accept: application/json; charset=UTF-8

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: ...

{
    "start": "http://localhost:7474/db/data/node/71",
    "end": "http://localhost:7474/db/data/node/70",
    "self": "http://localhost:7474/db/data/node/46",
    ...
}
Begin and commit a transaction in one request:

```
POST http://localhost:7474/db/data/transaction/commit HTTP/1.1
Accept: application/json; charset=UTF-8
Content-Type: application/json

{
  "statements": [
    {
      "statement": "CREATE (n) RETURN id(n)"
    }
  ]
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "results": [
    {
      "columns": ["id(n)"],
      "data": [
        {
          "row": [18]
        }
      ]
    }
  ],
  "errors": []
}
```
POST http://localhost:7474/db/data/transaction HTTP/1.1
Accept: application/json; charset=UTF-8
Content-Type: application/json

{
    "statements": [
        {
            "statement": "CREATE (n {props}) RETURN n",
            "parameters": {
                "props": {
                    "name": "My Node"
                }
            }
        }
    ]
}
REST API - Begin a Transaction

HTTP/1.1 201 Created
Content-Type: application/json
Location: http://localhost:7474/db/data/transaction/9

```json
{
  "commit": "http://localhost:7474/db/data/transaction/9/commit",
  "results": [ {
    "columns": [ "n" ],
    "data": [ {
      "row": [ { "name": "My Node" } ]
    } ]
  } ],
  "transaction": {"expires": "Tue, 10 Nov 2015 13:14:14 +0000"},
  "errors": [ ]
}
```
REST API - Use an Open Transaction

POST http://localhost:7474/db/data/transaction/9 HTTP/1.1
Accept: application/json; charset=UTF-8
Content-Type: application/json

{
    "statements" : [
        {
            "statement" : "CREATE (n) RETURN n"
        }
    ]
}
Rollback an open transaction:

Request:

DELETE http://localhost:7474/db/data/transaction/9 HTTP/1.1
Content-Type: application/json; charset=UTF-8

Response:

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "results" : [],
    "errors" : []
}
REST API - Handling Errors

**Request**

POST http://localhost:7474/db/data/transaction/commit HTTP/1.1
Accept: application/json; charset=UTF-8
Content-Type: application/json

```
{ "statements" : [{ "statement" : "Not a valid Statement." }]
}
```

**Response**

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

```
{ "results" : [],
 "errors" : [{
   "message" : "Invalid input 'T': expected <init> (line 1, column 1 (offset: 0))
   "Not a valid
Statement."\n\n^"
}
]
```
The service root contains the basic starting points for the database

GET http://localhost:7474/db/data/ HTTP/1.1
Content-Type: application/json; charset=UTF-8

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{  
  "extensions" : {},  
  "node" : "http://localhost:7474/db/data/node",  
  "node_index" : "http://localhost:7474/db/data/index/node",  
  "relationship_index" : "http://localhost:7474/db/data/index/relationship",  
  ...  
}
Overview

1. REST API
2. Neo4j in Java
3. Transactions
Embedding Neo4j in Java code

- You can use the Neo4j Core-Java-API directly
  - Package: org.neo4j.*

- or by using a driver that wraps that API
  For example:
  - Spring Data Neo4j
  - JDBC

Maven dependency

```xml
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j</artifactId>
      <version>2.3.1</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```
We create a class called EmbeddedNeo4j for our example

First we create relationship type by using an enum:

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

Next we defining some variables:

```java
GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;
```
In the next step we start the database server:

```java
graphDb = new GraphDatabaseFactory()
  .newEmbeddedDatabase( new File(DB_PATH) );
registerShutdownHook(graphDb);
```

We register a shutdown hook that will make sure the database shuts down when the JVM exits.

By design, all operations have to be performed in a transaction.

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

Syntactic sugar: try-with-resources
Here we create a small graph consisting of two nodes, connected with one relationship and some properties:

```java
try (Transaction tx = graphDb.beginTx()) {
    firstNode = graphDb.createNode();
    firstNode.setProperty("message", "Hello, ");
    secondNode = graphDb.createNode();
    secondNode.setProperty("message", "World!");

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

To shut down: `graphDb.shutdown();`
try ( Transaction tx = db.beginTx()) {
    Node myNode = db.createNode();
    myNode.setProperty( "name", "my node" );
    tx.success();
}

try ( Transaction ignored = db.beginTx();
    Result result = db.execute(  
    "match (n {name: 'my node'}) return n, n.name") ) {

    String output = "";
    while (result.hasNext()) {
        Map<String, Object> row = result.next();
        for (Entry<String, Object> column : row.entrySet()) {
            output += column.getKey() + " : "  
            + column.getValue() + " ; " ;
    }
}

output: n.name: my node; n: Node[0];
Although Neo4j is about graphs, the query language Cypher can be used with JDBC.

```xml
<dependencies>
  <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-jdbc</artifactId>
    <version>2.1.4</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>Neo4j releases</id>
    <url>http://m2.neo4j.org/content/repositories/releases</url>
  </repository>
...</repositories>
```
try (Connection conn = DriverManager.getConnection("jdbc:neo4j://localhost:7474", username, password)) {

String query =
"CREATE ({message:"Hello, \""}) -[:KNOWS {message:
"brave \""}]-> ({message:"world!\""})";
conn.createStatement().executeQuery(query);

ResultSet resultSet =
conn.createStatement().executeQuery("MATCH (a)-[r]->(b) "+ "RETURN a.message, r.message, b.message"");

while (resultSet.next()) {
    System.out.print(resultSet.getString("a.message"));
    System.out.print(resultSet.getString("r.message"));
    System.out.print(resultSet.getString("b.message"));
}
} catch (SQLException e1) {
    e1.printStackTrace();
}
Overview

1. REST API
2. Neo4j in Java
3. Transactions
Neo4j supports the ACID properties

- atomicity: If any part of a transaction fails, the database state is left unchanged
- consistency: Any transaction will leave 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
• All database operations must be performed in a transaction
• The default isolation level is READ_COMMITTED
  – Transactions will see data as soon as it has been committed and will not see data in other transactions that have not yet been committed
• Non-repeatable reads may occur (i.e., only write locks are acquired and held until the end of the transaction)
• Locks are acquired at the Node and Relationship level
• Deadlock detection is built into the core transaction management
The interaction cycle of working with transactions:
- Begin a transaction
- Perform database operations
- Mark the transaction as successful or not
  - Recall: `tx.success();`
- Finish the transaction
  - `tx.close();`

The transaction will not release the locks or memory it has acquired until it has been finished.
Example - Lost Updates

• Consider the following query:

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

• After run by 100 concurrent clients, what will be the value of the property `n.prop`?

• We can use locks to solve this problem:

```
MATCH (n:X {id: 42})
SET n._LOCK_ = true
SET n.prop = n.prop + 1
REMOVE n._LOCK_
```
Default Locking Behavior

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

- How to use the REST API from Java
- Neo4j Javadocs