eXtended Objects - Neo4j Datastore

Dirk Mahler

Table of Contents

This document describes the Neo4j datastore for eXtended Objects.

Introduction

As a graph database Neo4j provides very powerful capabilities to store and query highly interconnected data structures consisting of nodes and relationships between them. With release 2.0 the concept of labels has been introduced. One or more labels can be added to a single node:

```
create
  (a:Person:Actor)
set
  a.name="Harrison Ford"
```

Using labels it is possible to write comprehensive queries using Cypher:

```
match
  (a:Person)
where
  a.name="Harrison Ford"
return
  a.name;
```

If a node has a label it can be assumed that it represents some type of data which requires the presence of specific properties and relationships (e.g. property "name" for persons, "ACTED_IN" relations to movies). This implies that a Neo4j label can be represented as a Java interface and vice versa.

Person.java

```
@Label("Person") // The value "Person" can be omitted, in this case the class name is
used
public interface Person {
  String getName();
  void setName(String name);
}
```

Maven Dependencies

The Neo4j datastore for eXtended Objects is available from Maven Central and can be specified as dependency in pom.xml files:

pom.xml

```
<dependency>
    <!-- For using an embedded Neo4j instance -->
    <groupId>com.buschmais.xo</groupId>
    <artifactId>xo.neo4j.embedded</artifactId>
    <version>0.8.0</version>
</dependency>
<dependency>
    <!-- For using a remote Neo4j instance -->
    <groupId>com.buschmais.xo</groupId>
    <artifactId>xo.neo4j.remote</artifactId>
    <version>0.8.0</version>
</dependency>
<dependency>
    <!-- The XO API -->
    <groupId>com.buschmais.xo</groupId>
    <artifactId>xo.api</artifactId>
    <version>0.8.0</version>
</dependency>
<dependency>
    <!-- The XO runtime implementation -->
    <groupId>com.buschmais.xo</groupId>
    <artifactId>xo.impl</artifactId>
    <version>0.8.0</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <!-- An SLF4j binding -->
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <scope>runtime</scope>
    <version>1.7.21</version>
</dependency>
<!-- Optional dependencies for in-memory databases, i.e. URI "memory:///" -->
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-kernel</artifactId>
    <type>test-jar</type>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-io</artifactId>
    <type>test-jar</type>
    <version>3.1.2</version>
</dependency>
```

Bootstrapping

For a XOManagerFactory to be constructed a so called XO unit must be defined. There are two ways:

- Using a descriptor META-INF/xo.xml
- By using an instance of the class "com.buschmais.xo.XOUnit"

XOUnit via XML

An XO descriptor is a XML file located as classpath resource under "/META-INF/xo.xml" and defines one or more XO units. Each must be uniquely identified by a name. This is similar to the persistence unit approach defined by the Java Persistence API (JPA). The following snippet shows a minimum setups

META-INF/xo.xml (embedded database)

```
<v1:xo version="1.0" xmlns:v1="http://buschmais.com/xo/schema/v1.0">
    <xo-unit name="movies">
        <url>file:databases/movies</url>
        <provider>
com.buschmais.xo.neo4j.embedded.api.EmbeddedNeo4jXOProvider</provider>
        <types>
            <type>com.buschmais.xo.neo4j.doc.bootstrap.Person</type>
            <type>com.buschmais.xo.neo4j.doc.bootstrap.Actor</type>
        </types>
    </xo-unit>
</v1:xo>
```

META-INF/xo.xml (remote database)

```
<v1:xo version="1.0" xmlns:v1="http://buschmais.com/xo/schema/v1.0">
    <xo-unit name="movies">
        <url>bolt://localhost:7687</url>
        <provider>com.buschmais.xo.neo4j.remote.api.RemoteNeo4jXOProvider</provider>
        <types>
            <type>com.buschmais.xo.neo4j.doc.bootstrap.Person</type>
            <type>com.buschmais.xo.neo4j.doc.bootstrap.Actor</type>
        </types>
        <properties>
            <property name="neo4j.remote.username" value="neo4j"/>
            <property name="neo4j.remote.password" value="secret"/>
            <!--
            <property name="neo4j.remote.encryptionLevel" value="required"/>
            <property name="neo4j.remote.trust.strategy"
value="trustCustomCaSignedCertificates"/>
            <property name="neo4j.remote.trust.certificate" value="my.crt"/>
            <property name="neo4j.remote.statement.log.level" value="info"/>
            -->
        </properties>
    </xo-unit>
</v1:xo>
```

NOTE	Support for remote databases is experimental.

provider

The class name of the datastore provider

- `com.buschmais.xo.neo4j.embedded.api.EmbeddedNeo4jXOProvider` for embedded Neo4j databases
- `com.buschmais.xo.neo4j.remote.api.RemoteNeo4jXOProvider` for remote Neo4j databases via bolt protocol

url

The URL to pass to the Neo4j datastore. The following protocols are currently supported:

- "file:///C:/neo4j/movies": embedded local database using the specified directory as location for the Neo4j database
- "memory:///": embedded non-persistent in-memory database
- "bolt://localhost:7687": remote database

types

A list of all persistent types representing labels, relations or repositories

An XOManagerFactory instance can now be obtained as demonstrated in the following snippet:

```
XOManagerFactory movies = XO.createXOManagerFactory("movies");
XOManager xoManager = movies.createXOManager();

xoManager.currentTransaction().begin();

Person person = xoManager.create(Person.class);
person.setName("Indiana Jones");

xoManager.currentTransaction().commit();

xoManager.close();
movies.close();
```

XOUnit via XOUnitBuilder

It is also possible to create a XOManagerFactory using an instance of the class 'com.buschmais.xo.api.XOUnit':

```
XOUnit xoUnit = XOUnit.builder().provider(EmbeddedNeo4jXOProvider.class).uri(new URI(
"file:databases/movies")).type(Person.class).type(Actor.class)
        .build();
XOManagerFactory movies = XO.createXOManagerFactory(xoUnit);
XOManager xoManager = movies.createXOManager();

xoManager.currentTransaction().begin();

Person person = xoManager.create(Person.class);
person.setName("Indiana Jones");

xoManager.currentTransaction().commit();

xoManager.close();
movies.close();
```

Note: The class XOUnitBuilder provides a fluent interface for the parameters which may be specified for an XO unit.

Mapping Persistent Types

The Neo4j database provides the following native datastore concepts:

Node

An entity, e.g. a Person, Movie, etc. A node might have labels and properties.

Relationship

A directed relation between two nodes, might have properties. The lifecycle of relation depends

on the lifecycle of the nodes it connects.

The eXtended Objects datastore for Neo4j allows mapping of all these concepts to Java interfaces.

Nodes

Labeled Types

Neo4j allows adding one or more labels to a node. These labels are used by eXtended Objects to identify the corresponding Java type(s) a node is representing. Thus for each label that shall be used by the application a corresponding interface type must be created which is annotated with @Label.

Person.java

```
@Label
public interface Person {

    String getName();
    void setName(String name);

}
```

The name of the label defaults to the name of the interface, in this case 'Person'. A specific value can be enforced by adding a value to the @Label annotation.

It can also be seen that a label usually enforces the presence of specific properties (or relations) on a node. The name of a property - starting with a lower case letter - is used to store its value in the database, this can be overwritten using @Property. The following example demonstrates explicit mappings for a label and a property:

Person.java

```
@Label("MyPerson")
public interface Person {

    @Property("myName")
    String getName();
    void setName(String name);

}
```

The mapping of relations will be covered later.

Inheritance Of Labels

A labeled type can extend from one or more labeled types.

Actor.java

```
@Label
public interface Actor extends Person {
}
```

In this case a node created using the type Actor would be labeled with both 'Person' and 'Actor'. This way of combining types is referred to as 'static composition'.

Template Types

There might be situations where the same properties or relations shall be re-used between various labels. In this case template types can be used, these are just interfaces specifying properties and relations which shall be shared. The following example demonstrates how the property name of the labeled type Person is extracted to a template type:

Named.java

```
public interface Named {

    String getName();
    void setName(String name);

}
```

Person.java

```
@Label
public interface Person extends Named {

}
```

Relations

Unidirectional Relations

A node can directly reference other nodes using relation properties. A property of a labeled type or template type is treated as such if it references another labeled type or a collection thereof.

Movie.java

```
@Label
public interface Movie {

    String getTitle();
    void setTitle();

}
```

Actor.java

```
@Label
public interface Actor extends Person {

    List<Movie> getActedIn();

}
```

If no further mapping information is provided an outgoing unidirectional relation using the fully capitalized name of the property is assumed. The name may be specified using the @Relation annotation with the desired value. Furthermore using one of the annotations @Outgoing or @Incoming the direction of the relation can be specified.

Actor.java

```
@Label
public interface Actor extends Person {

    @Relation("ACTED_IN")
    @Outgoing
    List<Movie> getActedIn();

}
```

Note on multi-valued relations (i.e. collections):

- Only the following types are supported: 'java.util.Collection', 'java.util.List' or 'java.util.Set'.
- It is recommend to only specify the getter method of the property, as add or remove operations can be performed using the corresponding collection methods
- The provided 'java.util.Set' implementation ensures uniqueness of the relation to the referenced node, if this is not necessary 'java.util.List' should be prefered for faster add-operations.

Bidirectional Qualified Relations

Relations in many case shall be accessible from both directions. One possible way is to use two independent unidirectional relations which map to the same relation type; one of them annotated with @Outgoing, the other with @Incoming. There are some problems with this approach:

- it is not explicitly visible that the two relation properties are mapped to the same type
- renaming of the type or of one the properties might break the mapping

The recommended way is to use an annotation which qualifies the relation and holds the mapping information at a single point:

ActedIn.java

```
@Relation
@Retention(RUNTIME)
public @interface ActedIn {
}
```

Actor.java

```
@Label
public interface Actor extends Person {

    @ActedIn
    @Outgoing
    List<Movie> getActedIn();

}
```

Movie.java

```
@Label
public interface Movie {

    String getTitle();
    void setTitle();

    @ActedIn
    @Incoming
    List<Actor> getActors();
}
```

Typed Relations With Properties

If a relation between two nodes shall have properties a dedicated type must be declared and registered in the XOUnit. It must contain two properties returning the types of referenced types which are annotated with @Incoming and @Outgoing:

Directed.java

```
@Relation
public interface Directed {

    @Outgoing
    Director getDirector();

    @Incoming
    Movie getMovie();

    int getYear();
    void setYear(int year);

}
```

Director.java

```
@Label
public interface Director extends Person {

    List<Directed> getDirected();

}
```

Movie.java

```
@Label
public interface Movie {

    String getTitle();
    void setTitle();

    List<Directed> getDirected();

}
```

The relation is created explicitly:

```
xoManager.currentTransaction().begin();

Director director = xoManager.create(Director.class);
Movie movie = xoManager.create(Movie.class);

Directed directed = xoManager.create(director, Directed.class, movie);
directed.setYear(2017);

xoManager.currentTransaction().commit();
```

Note: If the typed relation references the same labeled type at both ends then the according properties of the latter must also be annotated with @Outgoing and @Incoming:

References.java

```
@Relation
public interface References {

    @Outgoing
    Movie getReferencing();

    @Incoming
    Movie getReferenced();

    int getMinute();
    void setMinute(int minute);

    int getSecond();
    void setSecond(int second);
}
```

Movie.java

```
@Label
public interface Movie {

    @Outgoing
    List<References> getReferenced();

    @Incoming
    List<References> getReferencedBy();

}
```

Typed relations may also be constructed using Template Types, i.e. types which define commonly used Properties.

Dynamic Properties

Labeled types or relation types may also define methods which execute a query on invocation and return the result:

Movie.java

```
@Label
public interface Movie {

    @ResultOf
    @Cypher("match (a:Actor)-[:ACTED_IN]->(m:Movie) where id(m)={this} return
count(a)")
    Long getActorCount();

    @ResultOf
    @Cypher("match (a:Actor)-[:ACTED_IN]->(m:Movie) where id(m)={this} and a.age={age}
return count(a)")
    Long getActorCountByAge(@Parameter("age") int age);

    @ActedIn
    @Incoming
    List<Actor> getActors();

}
```

Transient Properties

Properties of entities or relations can be declared as transient, i.e. they may be used at runtime but will not be stored in the database:

Person.java

```
@Label
public interface Person {

    @Transient
    String getName();
    void setName(String name);

}
```

User defined methods

It can be useful to provide a custom implementation of a method which has direct access to the underlying datatypes. This can be achieved using '@ImplementedBy', the following example uses an embedded Neo4j instance:

Person.java

```
@Label
public interface Person {

    @ImplementedBy(SetNameMethod.class)
    String setName(String firstName, String lastName);

    class SetNameMethod implements ProxyMethod<EmbeddedNode> {

        @Override
        public Object invoke(EmbeddedNode node, Object instance, Object[] args) {
            String firstName = (String) args[0];
            String lastName = (String) args[1];
            String fullName = firstName + " " + lastName;
            node.setProperty("name", fullName);
            return fullName;
        }
    }
}
```

Repositories

eXtended Objects supports the concepts of repositories. Similar to nodes and relationships they are also represented by Java interfaces and must be registered in the XOUnit. They allow the definition of dynamic properties:

PersonRepository.java

```
@Repository
public interface PersonRepository {

    @ResultOf
    @Cypher("MATCH (p:Person) WHERE p.name={name} RETURN p")
    Result<Person> getPersonsByName(@Parameter("name") String name);
}
```

A repository instance can be obtained from the XOManager:

```
xoManager.currentTransaction().begin();
Person person = xoManager.create(Person.class);
person.setName("Indiana Jones");

PersonRepository personRepository = xoManager.getRepository(PersonRepository.class);
Result<Person> personsByName = personRepository.getPersonsByName("Indiana Jones");
Person indy = personsByName.getSingleResult();

xoManager.currentTransaction().commit();
```

There is a predefined repository type that allows finding instances by labels. The following class declares an indexed property `name` for the label `Person`:

Person.java

```
@Label
public interface Person {

    @Indexed
    String getName();
    void setName(String name);

}
```

It is possible to define a repository which inherits from `TypedNeo4jRepository` and pass `Person` as type argument:

TypedPersonRepository.java

```
@Repository
public interface TypedPersonRepository extends TypedNeo4jRepository<Person> {
}
```

This allows using the provided `find` method:

```
xoManager.currentTransaction().begin();
Person person = xoManager.create(Person.class);
person.setName("Indiana Jones");

PersonRepository personRepository = xoManager.getRepository(PersonRepository.class);
Result<Person> personsByName = personRepository.getPersonsByName("Indiana Jones");
Person indy = personsByName.getSingleResult();

xoManager.currentTransaction().commit();
```