Neo4j OGM - An Object Graph Mapping Library for Neo4j

Version 2.0, 2016-05-25

Table of Contents

Preface

Introduction to Neo4j

What is a graph database?

A graph database is a storage engine that is specialised in storing and retrieving vast networks of information. It efficiently stores data as nodes and relationships and allows high performance retrieval and querying of those structures. Properties can be added to both nodes and relationships. Nodes can be labelled by zero or more labels, relationships are always directed and named.

Graph databases are well suited for storing most kinds of domain models. In almost all domains, there are certain things connected to other things. In most other modelling approaches, the relationships between things are reduced to a single link without identity and attributes. Graph databases allow to keep the rich relationships that originate from the domain equally well-represented in the database without resorting to also modelling the relationships as "things". There is very little "impedance mismatch" when putting real-life domains into a graph database.

About Neo4j

Neo4j *(http://neo4j.com/)* is an open source NOSQL graph database. It is a fully transactional database (ACID) that stores data structured as graphs consisting of nodes, connected by relationships. Inspired by the structure of the real world, it allows for high query performance on complex data, while remaining intuitive and simple for the developer.

Neo4j is very well-established. It has been in commercial development for 15 years and in production for over 12 years. Most importantly, it has an active and contributing community surrounding it, but it also:

- has an intuitive, rich graph-oriented model for data representation. Instead of tables, rows, and columns, you work with a graph consisting of nodes, relationships, and properties *(http://neo4j.com/docs/stable/graphdb-neo4j.html)*.
- has a disk-based, native storage manager optimised for storing graph structures with maximum performance and scalability.
- is scalable. Neo4j can handle graphs with many billions of nodes/relationships/properties on a single machine, but can also be scaled out across multiple machines for high availability.
- has a powerful graph query language called Cypher, which allows users to efficiently read/write data by expressing graph patterns.
- has a powerful traversal framework and query languages for traversing the graph.
- can be deployed as a standalone server, which is the recommended way of using Neo4j
- can be deployed as an embedded (in-process) database, giving developers access to its core Java API *(http://api.neo4j.org/)*

In addition, Neo4j provides ACID transactions, durable persistence, concurrency control, transaction recovery, high availability, and more. Neo4j is released under a dual free software/commercial licence model.

Querying the Graph with Cypher

Neo4j provides a graph query language called Cypher *(http://neo4j.com/docs/stable/cypher-query-lang.html)* which draws from many sources. It resembles SQL clauses but is centered around matching iconic representation of patterns in the graph.

Cypher queries typically begin with a `MATCH` clause, which can be used to provide a way to pattern match against the graph. Match clauses can introduce new identifiers for nodes and relationships. In the `WHERE` clause additional filtering of the result set is applied by evaluating expressions. The `RETURN`

clause defines which part of the query result will be available to the caller. Aggregation also happens in the return clause by using aggregation functions on some of the returned values. Sorting can happen in the `ORDER BY` clause and the `SKIP` and `LIMIT` parts restrict the result set to a certain window.

Cypher are executed against Neo4j Server using an HTTP based protocol which is utilised by Neo4j OGM.

Cypher Examples on the Movies Dataset

```
// Actors who acted in a Matrix movie:
MATCH (movie:Movie)<-[:ACTS_IN]-(actor)
WHERE movie.title =~ 'Matrix.*'
RETURN actor.name, actor.birthplace

// User-Ratings:
MATCH (user:User {login:'micha'})-[r:RATED]->(movie)
WHERE r.stars > 3
RETURN movie.title, r.stars, r.comment

// Mutual Friend recommendations:
MATCH (user:User {login:'micha'})-[:FRIEND]-(friend)-[r:RATED]->(movie)
WHERE r.stars > 3
RETURN friend.name, movie.title, r.stars, r.comment
```

Cypher Examples on the Movies Dataset

```
// Movie suggestions based on an actor:
MATCH (movie:Movie)<-[:ACTS_IN]-()-[:ACTS_IN]->(suggestion:Movie)
WHERE id(movie)=13
RETURN suggestion.title, count(*) ORDER BY count(*) DESC LIMIT 5

// Co-Actors, sorted by count and name of Lucy Liu
MATCH (lucy)-[:ACTS_IN]->(movie)<-[:ACTS_IN]-(co_actor)
WHERE lucy.name='Lucy Liu'
RETURN count(*), co_actor.name ORDER BY count(*) DESC, co_actor.name LIMIT 20

// Recommendations including counts, grouping and sorting
MATCH (:User {login:'micha'})-[:FRIEND]-()-[r:RATED]->(movie)
RETURN movie.title, avg(r.stars), count(*) ORDER BY avg(r.stars) DESC, count(*) DESC
```

Indexing

Neo4j's schema indexes are used automatically by Cypher when set up in your database. Neo4j OGM does not provide facilities for handling that setup out of the box. This is a seeding, migration or maintenance effort handled by the group responsible for the database maintenance.

Tutorial

The first part of this guide is a tutorial that takes the reader through steps necessary to get started with the Neo4j OGM. Neo4j OGM will power a web application that allows you to manage a College's Departments, Teaching Staff, Subjects, Students and Classes.

The complete source code for the application is available on Github *(https://github.com/neo4j-examples/neo4j-ogm-university)*.

Introduction

Neo4j OGM University is a demo application for the Neo4j OGM library.

It is a fully functioning web-application built using the following components

- Spring Boot
- Neo4j OGM
- Angular.js
- Twitter Bootstrap UI

The application's domain is a fictitious educational institution - Hilly Fields Technical College.

It leverages the power of Spring Boot and in particular the new Neo4j Object Graph mapping technology to provide a RESTful interface with which the web client interacts. The application is entirely stateless; every interaction involves a call to a Neo4j server, hopefully demonstrating the speed of the new technology, even over the wire.

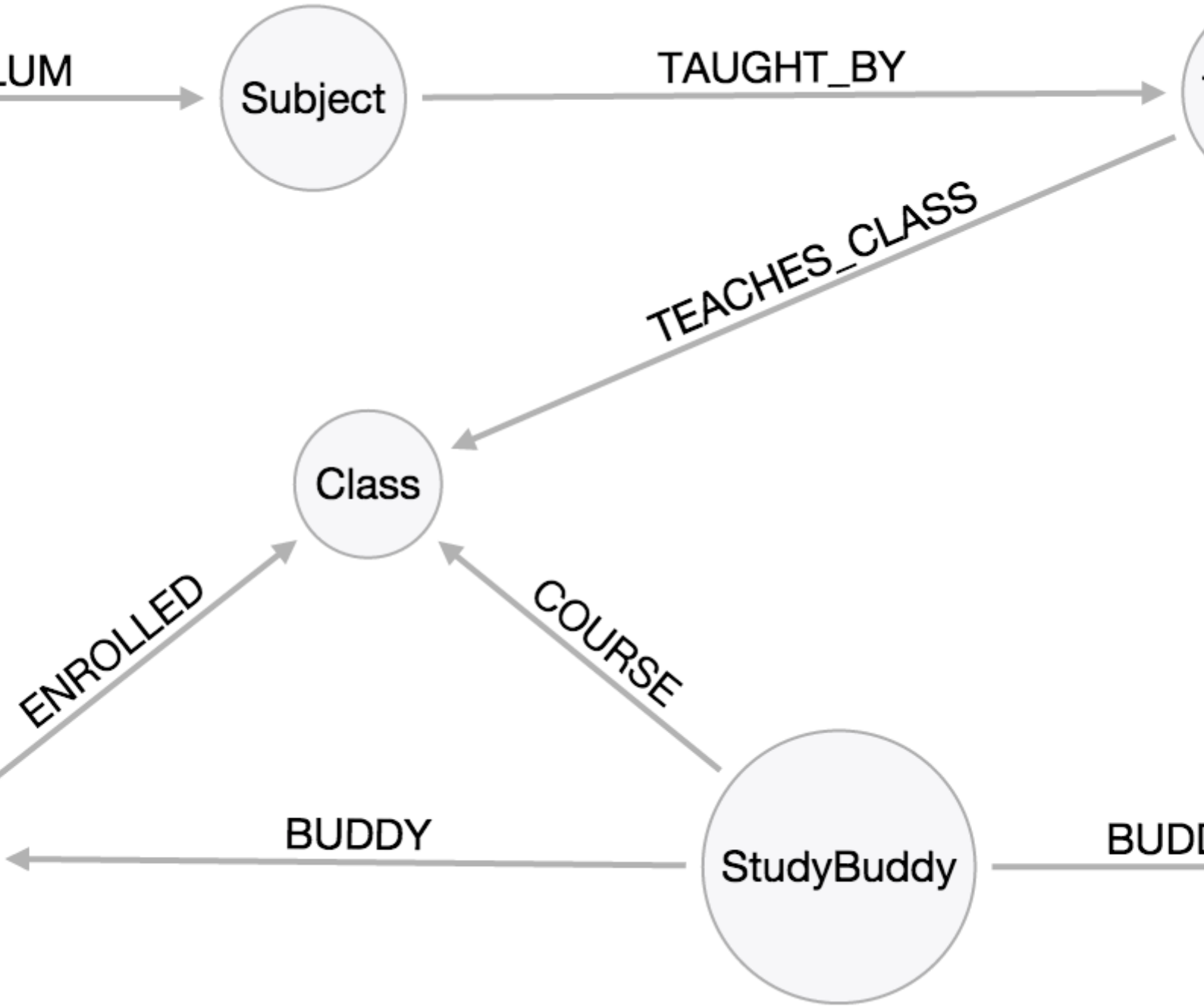
Domain Model

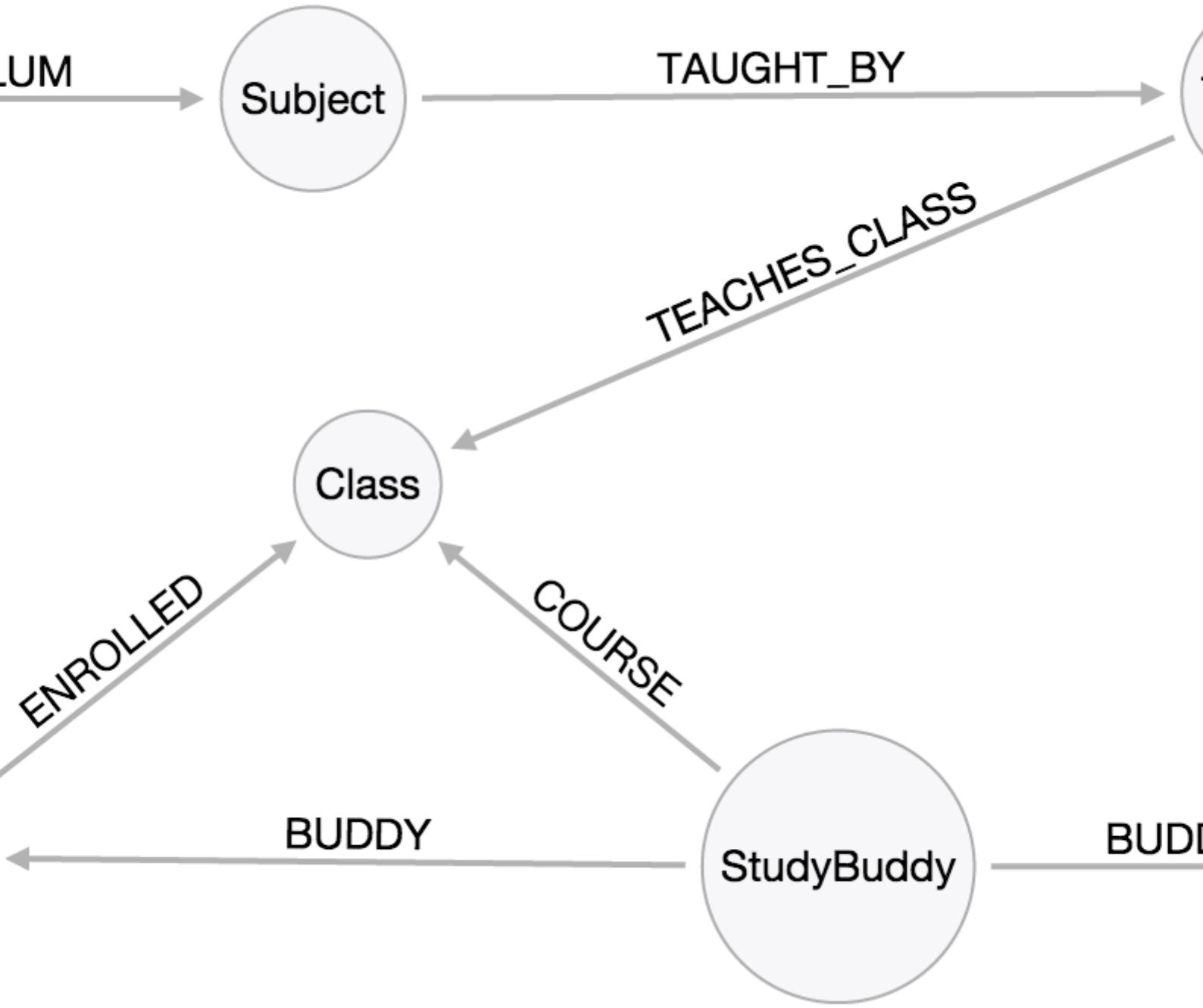
Before we get to any code, we want to whiteboard our graph model.

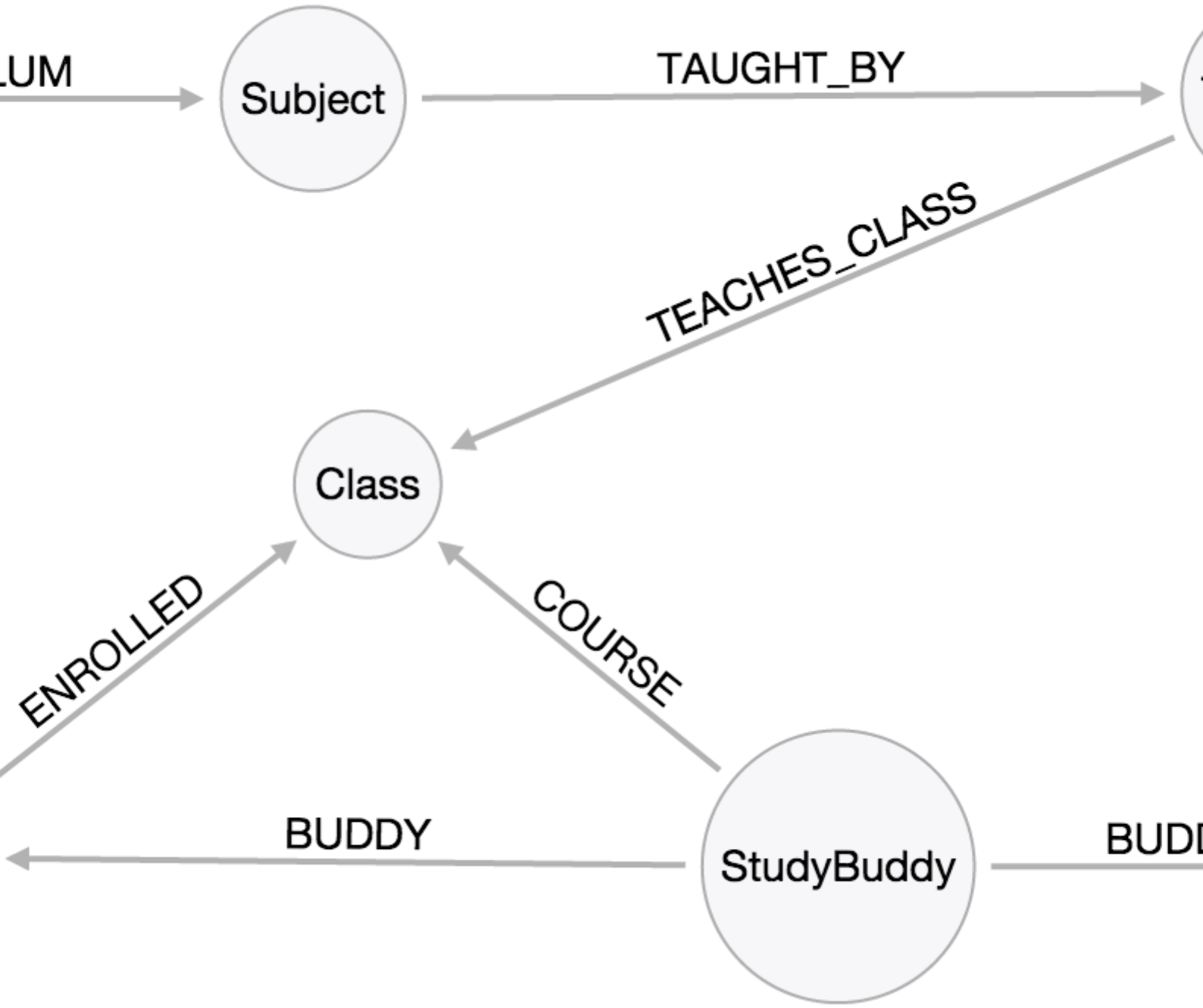
Our College will contain Departments, each of which offer various subjects taught by a teacher. Students enroll for courses or classes that teach a subject.

We're also going to model a Study Buddy which represents a group of students that get together to help one another study for a class.

Here's what we came up with.




LUM
Subject
TAUGHT_BY
TEACHES_CLASS
Class
ENROLLED
COURSE
BUDDY
StudyBuddy
BUDD



In Java, this is straightforward-

```
public class Department {
    private String name;
    private Set<Subject> subjects;
}

public class Subject {
    private String name;
    private Department department;
    private Set<Teacher> teachers;
    private Set<Course> courses;
}

public class Teacher {
    private String name;
    private Set<Course> courses;
    private Set<Subject> subjects;
}

public class Course {
    private String name;
    private Subject subject;
    private Teacher teacher;
    private Set<Enrollment> enrollments;
}

public class Student {
    private String name;
    private Set<Enrollment> enrollments;
    private Set<StudyBuddy> studyBuddies;
}

public class Enrollment {
    private Student student;
    private Course course;
    private Date enrolledDate;
}
```

Note that when a student enrolls for a course, we're going to keep track of the enrollment date. In the graph, this will be stored as a property on the ENROLLED relationship between a student and a course. This kind of rich relationship is managed by the class `Enrollment` and is known as a relationship entity.

Neo4j OGM

To simplify development, we're going to use Neo4j OGM, an object-graph mapping library. Much like JPA, we'll be annotating our POJOs and this will map them to nodes, relationships and properties in the graph.

Neo4j OGM 1.x works against server based installations of Neo4j and uses Cypher over the transactional HTTP endpoint. Neo4j OGM 2.x supports both server mode (HTTP and bolt) and embedded Neo4j.

Our sample application will use server mode with HTTP.

While the OGM takes care of boilerplate CRUD operations, it also provides us with the flexibility of writing our own Cypher queries and controlling persistence depth as we shall see later in this tutorial.

Annotations

Neo4j OGM supports mapping annotated and non-annotated objects models. It's possible to save any POJO with the exception of a Relationship Entity without annotations to the graph. The framework will then apply conventions to decide what to do.

Maven Dependency

Before we can use any Neo4j OGM annotations, we need to add the maven dependency.

Maven dependencies for Neo4j OGM 2.x

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-core</artifactId>
    <version>{version}</version>
</dependency>
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-http-driver</artifactId>
    <version>{version}</version>
</dependency>
```

Refer to Setup for Gradle or Ivy dependencies.

Note that the dependencies for Neo4j OGM 1.x differ -

Maven dependencies for Neo4j OGM 1.x

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm</artifactId>
    <version>{version}</version>
</dependency>
```

Nodes

POJOs annotated with `@NodeEntity` will be represented as nodes in the graph. The label assigned to this node can be specified via the `label` property on the annotation; if not specified, it will default to the simple class name of the entity. Each parent class in addition also contributes a label to the entity (with the exception of java.lang.Object). This is useful when we want to retrieve collections of super types.

Let's go ahead and annotate all our node entities. Note that we're overriding the default label for a `Course` with `Class`

```
@NodeEntity
public class Department {
    private String name;
    private Set<Subject> subjects;
}

@NodeEntity
public class Subject {
    private String name;
    private Department department;
    private Set<Teacher> teachers;
    private Set<Course> courses;
}

@NodeEntity
public class Teacher {
    private String name;
    private Set<Course> courses;
    private Set<Subject> subjects;
}

@NodeEntity(label="Class")
public class Course {
    private String name;
    private Subject subject;
    private Teacher teacher;
    private Set<Enrollment> enrollments;
}

@NodeEntity
public class Student {
    private String name;
    private Set<Enrollment> enrollments;
    private Set<StudyBuddy> studyBuddies;
}
```

Relationships

Next up, the relationships between the nodes. Every field in an entity that references another entity is backed by a relationship in the graph. The `@Relationship` annotation allows you to specify both the type of the relationship and the direction. By default, the direction is assumed to be `OUTGOING` and the type is the UPPER_SNAKE_CASE field name. We're going to be specific about the relationship type to avoid using the default and also make it easier to refactor classes later by not being dependent on the field name.

```
@NodeEntity
public class Department {
    private String name;

    @Relationship(type = "CURRICULUM")
    private Set<Subject> subjects;
}

@NodeEntity
public class Subject {
    private String name;

    @Relationship(type="CURRICULUM", direction = Relationship.INCOMING)
    private Department department;

    @Relationship(type = "TAUGHT_BY")
    private Set<Teacher> teachers;

    @Relationship(type = "SUBJECT_TAUGHT", direction = "INCOMING")
    private Set<Course> courses;
}

@NodeEntity
public class Teacher {
    private String name;

     @Relationship(type="TEACHES_CLASS")
     private Set<Course> courses;

     @Relationship(type="TAUGHT_BY", direction = Relationship.INCOMING)
     private Set<Subject> subjects;
}

@NodeEntity(label="Class")
public class Course {
    private String name;

     @Relationship(type= "SUBJECT_TAUGHT")
     private Subject subject;

     @Relationship(type= "TEACHES_CLASS", direction=Relationship.INCOMING)
     private Teacher teacher;

     @Relationship(type= "ENROLLED", direction=Relationship.INCOMING)
     private Set<Enrollment> enrollments = new HashSet<>();
}

@NodeEntity
public class Student {
    private String name;

    @Relationship(type = "ENROLLED")
    private Set<Enrollment> enrollments;

    @Relationship(type = "BUDDY", direction = Relationship.INCOMING)
    private Set<StudyBuddy> studyBuddies;
}
```

Relationship Entities

We have one more entity, and that is the `Enrollment`. As discussed earlier, this is a relationship entity since it manages the underlying `ENROLLED` relation between a student and course. It isn't a simple relation because it has a relationship property called `enrolledDate`.

A relationship entity must be annotated with `@RelationshipEntity` and also the type of relationship. In this case, the type of relationship is `ENROLLED` as specified in both the `Student` and `Course` entities. We are also going to indicate to the OGM the start and end node of this relationship.

```
@RelationshipEntity(type = "ENROLLED")
public class Enrollment {

    private Long id;

    @StartNode
    private Student student;

    @EndNode
    private Course course;

    private Date enrolledDate;

}
```

@GraphId

Every node and relationship persisted to the graph has an id. Neo4j OGM uses this to identify and re-connect the entity to the graph. Specifying a Long id field is required. If such a field exists on the entity, then Neo4j OGM will use it automatically. Otherwise, a Long field must be annotated with `@GraphId`.

Since every entity requires an id, we're going to create an `Entity` superclass. This is an abstract class, so you'll see that the nodes do not inherit an `Entity` label, which is exactly what we want.

Also observe the null checks for the `id` field in the `equals` and `hashCode`. This is required because the `id` is null till the entity is persisted to the graph.

```
public abstract class Entity {

    private Long id;

    public Long getId() {
        return id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || id == null || getClass() != o.getClass()) return false;

        Entity entity = (Entity) o;

        if (!id.equals(entity.id)) return false;

        return true;
    }

    @Override
    public int hashCode() {
        return (id == null) ? -1 : id.hashCode();
    }
}
```

Our entities will now extend this class, for example

```
@NodeEntity
public class Department extends Entity {
    private String name;

    @Relationship(type = "CURRICULUM")
    private Set<Subject> subjects;

    public Department() {

    }
}
```

The OGM also requires an public no-args constructor to be able to construct objects, we'll make sure all our entities have one.

Converters

Neo4j supports numerics, Strings, booleans and arrays of these as property values. How do we handle the `enrolledDate` since `Date` is not a valid data type? Luckily for us, Neo4j OGM provides many converters out of the box, one of which is a Date to Long converter. We simply annotate the field with `@DateLong` and the conversion of the Date to it's Long representation and back is handled by the OGM when persisting and loading from the graph.

```
@RelationshipEntity(type = "ENROLLED")
public class Enrollment {

    private Long id;

    @StartNode
    private Student student;

    @EndNode
    private Course course;

    @DateLong
    private Date enrolledDate;

}
```

Configuration

Since the OGM 2.x works both against a remote server as well as embedded Neo4j, we'll need to configure it to use the appropriate driver. We'll allow the OGM to auto-configure itself by providing a file called `ogm.properties` in the classpath.

```
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://username:password@localhost:7474
```

Here, we're setting up the HTTP Driver to connect to our local Neo4j server.

OGM 1.x does not require this configuration. Instead, the URL to the Neo4j server is provided to the `Session` described below.

Session

So our domain entities are annotated, now we're ready persist them to the graph!

The smart object mapping capability is provided by the `Session`. A `Session` is obtained from a `SessionFactory`.

We're going to set up the `SessionFactory` just once and have it produce as many sessions as required.

```
public class Neo4jSessionFactory {

    private final static SessionFactory sessionFactory = new SessionFactory("school.domain");
    private static Neo4jSessionFactory factory = new Neo4jSessionFactory();

    public static Neo4jSessionFactory getInstance() {
        return factory;
    }

    private Neo4jSessionFactory() {
    }

    public Session getNeo4jSession() {
        return sessionFactory.openSession();
    }
}
```

OGM 1.x needs to provide the URL to the Neo4j Server to the `openSession` method like this

```
    sessionFactory.openSession("http://localhost:7474");
```

The `SessionFactory` constructor accepts packages that are to be scanned for domain metadata. The domain objects in our university application are grouped under `school.domain`. When the `SessionFactory` is created, it will scan `school.domain` for potential domain classes and construct the object mapping metadata to be used by all sessions created thereafter.

A `Session` requires a URL of the remote Neo4j database. All operations within the session will be performed against this remote database. The `Session` keeps track of changes made to entities and relationships and persists ones that have been modified on save. However, when loading an entity, it always hits the database and never returns cached objects. This is why the life of the session is important. For the purpose of this demo application, we'll be refreshing the session frequently and hence our session is going to be long lived.

Our university application will use the following operations

```
public interface Service<T> {

    Iterable<T> findAll();

    T find(Long id);

    void delete(Long id);

    T createOrUpdate(T object);

}
```

These CRUD interactions with the graph are all handled by the `Session`. We wrote `GenericService` to deal with `Session` operations.

```
public abstract class GenericService<T> implements Service<T> {

 private static final int DEPTH_LIST = 0;
 private static final int DEPTH_ENTITY = 1;
 private Session session = Neo4jSessionFactory.getInstance().getNeo4jSession();

 @Override
 public Iterable<T> findAll() {
     return session.loadAll(getEntityType(), DEPTH_LIST);
 }

 @Override
 public T find(Long id) {
     return session.load(getEntityType(), id, DEPTH_ENTITY);
 }

 @Override
 public void delete(Long id) {
     session.delete(session.load(getEntityType(), id));
 }

 @Override
 public T createOrUpdate(T entity) {
     session.save(entity, DEPTH_ENTITY);
     return find(((Entity) entity).getId());
 }

 public abstract Class<T> getEntityType();
}
```

One of the features of Neo4j OGM is variable depth persistence. This means you can vary the depth of fetches depending on the shape of your data and application. The default depth is 1, which loads simple properties of the entity and its immediate relations. This is sufficient for the `find` method, which is used in the application to present a create or edit form for an entity.

Loading relationships is not required however when listing all entities of a type. We merely require the id and name of the entity, and so a depth of 0 is used by `findAll` to only load simple properties of the entity but skip its relationships.

The default save depth is -1, or everything that has been modified and can be reached from the entity up to an infinite depth. This means we can persist all our changes in one go.

This `GenericService` takes care of CRUD operations for all our entities! All we did was delegate to the `Session`; no need to write persistence logic for every entity.

Queries

Popular Study Buddies is a report that lists the most popular peer study groups. This requires a custom Cypher query. It is easy to supply a Cypher query to the `query` method available on the `Session`.

```
Service("studyBuddyService")
public class StudyBuddyServiceImpl extends GenericService<StudyBuddy> implements StudyBuddyService {

    @Override
    public Iterable<Map<String,Object>> getStudyBuddiesByPopularity() {
        String query =
            "MATCH (s:StudyBuddy)<-[:BUDDY]-(p:Student) return p, count(s) as buddies ORDER BY buddies
DESC";

        return Neo4jSessionFactory.getInstance().getNeo4jSession()
                .query(query, Collections.EMPTY_MAP);
    }

    @Override
    public Class<StudyBuddy> getEntityType() {
        return StudyBuddy.class;
    }
}
```

The `query` provided by the `Session` can return a domain object, a collection of them, or a `org.neo4j.ogm.model.Result`.

Conclusion

With not much effort, we've built all the services that tie together this application. All that is required is adding controllers and building the UI. The fully functioning application is available at Github *(https://github.com/neo4j-examples/neo4j-ogm-university)*.

We encourage you to read the reference guide that follows and apply the concepts learned by forking the application and adding to it.

Reference Documentation

This part of the guide provides the reference documentation for the Neo4j OGM.

Its content covers information about the programming model, APIs, concepts, annotations and technical details of the Neo4j OGM.

Whenever you look for the means to employ the full power of the OGM library, you should be able to find your answers in this reference section. If you don't, please inform us about missing or incorrect content.

About the Neo4j OGM Library

Neo4j OGM is a fast object-graph mapping library for Neo4j, optimised for server-based installations and utilising Cypher via the transactional HTTP Endpoint. Focused on performance, it introduces a number of innovations, including non-reflection based classpath scanning for much faster startup times; variable-depth persistence to allow you to fine-tune requests according to the characteristics of your graph; smart object-mapping to reduce redundant requests to the database, improve latency and minimise wasted CPU cycles; and user-definable session lifetimes, helping you to strike a balance between memory-usage and server request efficiency in your applications.

Neo4j OGM aims to simplify development with the Neo4j graph database. Like JPA, it uses annotations on simple POJO domain objects. Together with metadata, the annotations drive mapping the POJO entities and their fields to nodes, relationships, and properties in the graph database.

Compatibility

The following table lists compatibility of the OGM with different versions of Neo4j.

Neo4j-OGM Version	Neo4j Version
2.0.2	2.3.x, 3.0.x
2.0.1	2.2.x, 2.3.x
1.1.5	2.1.x, 2.2.x, 2.3.x

Multiple driver implementations

As of 2.0, the OGM provides support for connecting to Neo4j using different drivers. The following drivers are available.

- Http driver
- Embedded driver
- Bolt driver

Please see the Setup Guide for more details about how to configure the OGM to use the driver you require.

Cypher

Cypher is Neo4j's powerful query language. It is understood by all the different drivers in the OGM which means that your application code should run identically, whichever driver you choose to use. This makes application development much easier: you can use the Embedded Driver for your integration tests, and then plugin the Http Driver or the Bolt Driver when deploying your code into a production client-server environment.

Design Considerations

The OGM attempts to minimise the Cypher payload when persisting your objects to the graph. This is important for two reasons. Firstly in client-server mode, every network interaction involves an overhead (both bandwidth but more so latency) which impacts the response times of your application. Secondly, requests containing redundant operations (such as updating an object which hasn't changed) are unnecessary, and have similar impacts. We have approached this problem in a number of ways:

Variable-depth persistence

You can now tailor your persistence requests according to the characteristics of the portions of your graph you want to work with. This means you can choose to make deeper or shallower fetches based on fine tuning the types and amounts of data you want to transfer based on your individual constraints.

If you know that you aren't going to need an object's related objects, you can choose not to fetch them by specifying the fetch-depth as 0. Alternatively if you know that you will always want to a person's complete set of friends-of-friends, you can set the depth to 2.

Smart object-mapping

Neo4j OGM introduces smart object-mapping. This means that all other things being equal, it is possible to reliably detect which nodes and relationships needs to be changed in the database, and which don't.

Knowing what needs to be changed means we don't need to flood Neo4j with requests to update objects that don't require changing, or create relationships that already exist. We can minimise the amount of data we send across the wire as a result, which results in a faster network interaction, and fewer CPU cycles consumed on the server.

User-definable Session lifetime

Supporting the smart object-mapping capability is the `Session` whose lifetime can be managed in code. For example, associated with single *fetch-update-save* cycle or unit of work.

The advantage of longer-running sessions is that you will be able to make more efficient requests to the database at the expense of the additional memory associated with the session. The advantage of shorter sessions is they impose almost no overhead on memory, but will result in less efficient requests to Neo4j when storing and retrieving data.

Overview

The basic concepts of the Object-Graph Mapping (OGM) library are explained in this chapter.

Getting Started

To get started with a simple application, you need only your domain model and (optionally) the annotations provided by the library. You use annotations to mark domain objects to be reflected by nodes and relationships of the graph database. For individual fields the annotations allow you to declare how they should be processed and mapped to the graph. For property fields and references to other entities this is straightforward.

Adding Neo4j Graph Queries

To use advanced functionality like Cypher queries, a basic understanding of the graph data model is required. The graph data model is explained in the chapter about Neo4j.

Managing Relationships

Relationships between entities are first class citizens in a graph database and therefore worth a separate chapter describing their usage in Neo4j OGM.

Session

Neo4j OGM offers a Session for interacting with the mapped entities and the Neo4j graph database.

Mapping Strategies

Because Neo4j is a schema-free database, the OGM uses a simple mechanism to map Java types to Neo4j nodes using labels. How that works is explained here: Entity Type Representation.

Transactional Support

Neo4j uses transactions to guarantee the integrity of your data and Neo4j OGM supports this fully. The implications of this are described here.

Setup

Neo4j OGM dramatically simplifies development, but some setup is required. For building the application, your build automation tool needs to be configured to include the Neo4j OGM dependencies and after the build setup is complete, the application needs to be configured to make use of Neo4j OGM.

Neo4j OGM projects can be built using Maven, Gradle or Ant/Ivy.

Dependencies for the Neo4j OGM

The OGM dependencies consist of *neo4j-ogm-core*, together with the relevant dependency declarations on the drivers you want to use in production. If you're not using a particular driver, you don't need to declare it.

Maven dependencies for Neo4j OGM 2.x

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-core</artifactId>
    <version>2.0</version>
</dependency>

<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-http-driver</artifactId>
    <version>2.0</version>
</dependency>

<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-embedded-driver</artifactId>
    <version>2.0</version>
</dependency>

<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-bolt-driver</artifactId>
    <version>2.0</version>
</dependency>
```

Gradle dependencies for Neo4j OGM 2.x

```
dependencies {
    compile 'org.neo4j:neo4j-ogm-core:2.0'
    compile 'org.neo4j:neo4j-ogm-http-driver:2.0'
    compile 'org.neo4j:neo4j-ogm-embedded-driver:2.0'
    compile 'org.neo4j:neo4j-ogm-bolt-driver:2.0'
}
```

Ivy dependencies for Neo4j OGM 2.x

```
<dependency org="org.neo4j" name="neo4j-ogm-core" rev="2.0"/>
<dependency org="org.neo4j" name="neo4j-ogm-http-driver" rev="2.0"/>
<dependency org="org.neo4j" name="neo4j-ogm-embedded-driver" rev="2.0"/>
<dependency org="org.neo4j" name="neo4j-ogm-bolt-driver" rev="2.0"/>
```

Maven dependencies for Neo4j OGM 1.x

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm</artifactId>
    <version>2.0</version>
</dependency>
```

Gradle dependencies for Neo4j OGM 1.x

```
dependencies {
    compile 'org.neo4j:neo4j-ogm:2.0'
}
```

Ivy dependencies for Neo4j OGM 1.x

```
<dependency org="org.neo4j" name="neo4j-ogm" rev="2.0"/>
```

OGM 2.x Configuration

In previous versions, you could only connect to the database over HTTP. 2.0 now provides support for connecting to Neo4j by configuring one of the following Drivers:

- HTTP driver
- Embedded driver
- Bolt driver

You must declare the driver(s) you want to use in your pom. See the dependencies section for more information.

There are two basic ways to supply this configuration information: via a single properties file, or programmatically.

Properties file Configuration

Unless you supply an explicit Configuration object to the SessionFactory (see below), the OGM will attempt to auto-configure itself using a file called `ogm.properties`, which it expects to find on the classpath. If you want to configure the OGM using a properties file, but with a *different* filename, you must set a System property or Environment variable called 'ogm.properties' pointing to the alternative configuration file you want to use.

Java Configuration

In some cases you won't want to, or will not be able to provide configuration information using a properties file. In this case you can configure the OGM programmatically instead.

The Configuration object provides a fluent API to set various configuration options.

The following sections describe how to configure the OGM driver using either a properties file or via Java Configuration.

Driver Configuration

Configuring the HTTP Driver

The HTTP Driver connects to and communicates with a Neo4j server via HTTP. If your application is running in client-server mode, you must use either the HTTP or Bolt driver.

ogm.properties

```
driver=org.neo4j.ogm.drivers.http.driver.HttpDriver
URI=http://user:password@localhost:7474
```

Java Configuration

```
Configuration configuration = new Configuration()
            .driverConfiguration()
            .setDriverClassName("org.neo4j.ogm.drivers.http.driver.HttpDriver")
            .setURI("http://user:password@localhost:7474")

// having created a new configuration object, we pass it as the first argument to a SessionFactory
instance
new SessionFactory(configuration, packages...);
```

Note: Please see the section below describing the different ways you can pass credentials to the HTTP Driver

Configuring the Bolt Driver

The Bolt Driver connects to and communicates with a Neo4j server via the binary Bolt protocol. If your application is running in client-server mode, you must use either the HTTP or Bolt driver.

ogm.properties

```
#Driver, required
driver=org.neo4j.ogm.drivers.bolt.driver.BoltDriver

#URI of the Neo4j database, required. If no port is specified, the default port 7687 is used. Otherwise, a
port can be specified with bolt://neo4j:password@localhost:1234
URI=bolt://neo4j:password@localhost

#Connection pool size (the maximum number of sessions per URL), optional, defaults to 50
connection.pool.size=150

#Encryption level (TLS), optional, defaults to REQUIRED. Valid values are NONE,REQUIRED
encryption.level=NONE

#Trust strategy, optional, not used if not specified. Valid values are
TRUST_ON_FIRST_USE,TRUST_SIGNED_CERTIFICATES
trust.strategy=TRUST_ON_FIRST_USE

#Trust certificate file, required if trust.strategy is specified
trust.certificate.file=/tmp/cert
```

Java Configuration

```
Configuration configuration = new Configuration();
               configuration.driverConfiguration()
               .setDriverClassName("org.neo4j.ogm.drivers.bolt.driver.BoltDriver")
               .setURI("bolt://neo4j:password@localhost")
               .setEncryptionLevel("NONE")
               .setTrustStrategy("TRUST_ON_FIRST_USE")
               .setTrustCertFile("/tmp/cert");


// having created a new configuration object, we pass it as the first argument to a SessionFactory
instance

new SessionFactory(configuration, packages...);
```

Note: Please see the section below describing the different ways you can pass credentials to the HTTP/Bolt Drivers

Configuring the Embedded Driver

The Embedded Driver connects directly to the Neo4j database engine. There is no server involved, therefore no network overhead between your application code and the database. You should use the Embedded driver if you don't want to use a client-server model, or if your application is running as a Neo4j Unmanaged Extension. You can specify a permanent data store location to provide durability of your data after your application shuts down, or you can use an impermanent data store, which will only exist while your application is running.

ogm.properties (permanent data store)

```
driver=org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver
URI=file:///var/tmp/neo4j.db
```

ogm.properties (impermanent data store)

```
driver=org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver
```

Java Configuration (permanent data store)

```
Configuration configuration = new Configuration()
            .driverConfiguration()
            .setDriverClassName("org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver")
            .setURI("file://home/bilbo");
```

Java Configuration (impermanent data store)

```
Configuration configuration = new Configuration()
            .driverConfiguration()
            .setDriverClassName("org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver")
```

As you can see to use an impermanent data store, you just omit the URI attribute.

Configuring the Embedded Driver in an Unmanaged Extension

When your application is running as unmanaged extension inside the Neo4j server itself, you will need to set up the Driver configuration slightly differently. In this situation, an existing `GraphDatabaseService` will already be available via a `@Context` annotation, and you must configure the Components framework to enable the OGM to use the provided instance. Note your application should typically do this only once.

```
    Components.setDriver(new EmbeddedDriver(graphDatabaseService));
```

Credentials

If you are using the HTTP or Bolt Driver you have a number of different ways to supply credentials to the Driver Configuration.

ogm.properties:

```
# embedded
URI=http://user:password@localhost:7474

# separately
username="user"
password="password"
```

Java Configuration

```
// embedded
Configuration configuration = new Configuration()
             .driverConfiguration()
             .setURI("bolt://user:password@localhost");

// separately as plain text
Configuration configuration = new Configuration()
             .driverConfiguration()
             .setCredentials("user", "password);

// using a Credentials object
Credentials credentials = new UsernameAndPasswordCredentials("user", "password");
Configuration configuration = new Configuration()
             .driverConfiguration()
             .setCredentials(credentials);
```

Note: Currently only Basic Authentication is supported by Neo4j, so the only Credentials implementation supplied by the OGM is `UsernameAndPasswordCredentials`

Testing

In 2.0, the `Neo4jIntegrationTestRule` class has been removed from the test-jar.

In previous versions this class provided access to an underlying `GraphDatabaseService` instance, allowing you to independently verify your code was working correctly. However it is incompatible with the Driver interfaces in 2.0, as it always requires you to connect using HTTP.

The recommended approach is to configure an Embedded Driver for testing as described above, although you can still use an in-process HTTP server if you wish (see below). Please note that if you're just using the Embedded Driver for your tests you do not need to include any additional test jars in your pom.

Log levels

When running unit tests, it can be useful to see what the OGM is doing, and in particular to see the Cypher requests being transferred between your application and the database. The OGM uses `slf4j` along with `Logback` as its logging framework and by default the log level for all the OGM components is set to WARN, which does not include any Cypher output. To change the OGM log level, create a file **logback-test.xml** in your test resources folder, configured as shown below:

logback-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d %5p %40.40c:%4L - %m%n</pattern>
        </encoder>
    </appender>

    <!--
      ~ Set the required log level for the OGM components here.
      ~ To just see Cypher statements set the level to "info"
      ~ For finer-grained diagnostics, set the level to "debug".
    -->
    <logger name="org.neo4j.ogm" level="info" />

    <root level="warn">
        <appender-ref ref="console" />
    </root>

</configuration>
```

Production

In production, you can set the log level in exactly the same way, but the file should be called **logback.xml**, not **logback-test.xml**. Please see the <<http://logback.qos.ch/manual/,Logback manual>> for further details.

Using an in-process server for testing

If you want don't want to use the Embedded Driver to run your tests, it is still possible to create an in-process HTTP server instead. Just like the Embedded Driver, a TestServer exposes a GraphDatabaseService instance which you can use in your tests. You should always close the server when you're done with it.

You'll first need to add the OGM test-jar dependency to your pom:

```
        <dependency>
           <groupId>org.neo4j</groupId>
           <artifactId>neo4j-ogm-test</artifactId>
           <version>2.0</version>
           <type>test-jar</type>
           <scope>test</scope>
        </dependency>
```

Next, create a TestServer instance:

```
      testServer = new TestServer.Builder()
                       .enableAuthentication(true)    // defaults to false
                       .transactionTimeoutSeconds(10) // defaults to 30 seconds
                       .port(2222)                    // defaults to a random non-privileged port
                       .build();
```

A TestServer is backed by an impermanent database store, and configures the OGM to use an HttpDriver. The driver authenticates automatically if you have requested an authenticating server so you don't have to do provide additional credentials.

Example test class using an in-process HTTP server

```
private static TestServer testServer;

@BeforeClass
public static setupTestServer() {
      testServer = new TestServer.Builder().build();
}

@AfterClass
public static teardownTestServer() {
    testServer.close();
}

@Test
public void shouldCreateUser() {

    session.save(new User("Bilbo Baggins"));

    GraphDatabaseService db = testServer.getGraphDatabaseService();
    try (Transaction tx = db.beginTx()) {
        Result r = db.execute("MATCH (u:User {name: 'Bilbo Baggins'}) RETURN u");
        assertTrue(r.hasNext());
        tx.success();
    }
}
```

Migrating from OGM 1.x to 2.x

OGM 2.0 introduces a few minor changes that you will need to take into account when migrating an existing 1.x application. These changes are a consequence of the support for different database drivers. In 1.x, the only connectivity to Neo4j was over HTTP, and the code reflected this in its design, as it closely coupled the session with the HTTP client. In 2.0 this design is no longer appropriate, and the connection to the database is abstracted via a Driver interface.

This has an impact on your application code in two areas, testing (discussed above) and session configuration.

Session Configuration differences between 1.x and 2.x

In 2.0, the SessionFactory API has been considerably simplified. There is now only one method to open a session: `openSession()`. You can no longer pass in any credentials or other attributes as arguments: this information is now part of the Configuration as discussed above.

On the other hand, there are now two ways to create a SessionFactory. You can continues to use the default constructor, in which case the SessionFactory will be auto-configured from a configuration properties file. Alternatively you can supply an explicit Configuration object to the constructor.

Example: Auto-configured session

An auto-configured session requires that you set up a properties-based configuration file, as described earlier. You can then simply instantiate a SessionFactory in the usual way, passing in the domain class packages to the constructor.

```
SessionFactory sessionFactory = new SessionFactory("org.neo4j.example.domain");
Session session = sessionFactory.openSession()
```

Example: Explicitly-configured session

If you want to explicitly configure the SessionFactory you must supply a Configuration object as the first argument to the constructor, followed by the domain class packages.

```
Configuration configuration = new Configuration()
            .driverConfiguration()
            .setDriverClassName("org.neo4j.ogm.drivers.http.driver.HttpDriver")
            .setURI("http://localhost:7474")
            .setCredentials("user", "password")

SessionFactory sessionFactory = new SessionFactory(configuration, "org.neo4j.example.domain");
Session session = sessionFactory.openSession();
```

Refer to the Java Configuration section above for more details about the various configuration options.

OGM 1.x Configuration

Driver configuration

Note: OGM 1.x only supports Http (server-based) connectivity to Neo4j. If you want to use an Embedded database to connect with a Neo4j server, you'll need to upgrade to OGM 2.0

If you're running against Neo4j 2.2 or later and authentication is enabled, you will need to supply connection credentials. This can be accomplished by supplying the username and password as parameters to the `SessionFactory.openSession` method, or by embedded them into the URL such as `http://username:password@localhost:7474`.

Passing connection credentials when opening the session

```
SessionFactory sessionFactory = new SessionFactory("org.neo4j.example.domain");
Session session = sessionFactory.openSession("http://localhost:7474", username, password);
```

Embedding connection credentials in the URL

```
SessionFactory sessionFactory = new SessionFactory("org.neo4j.example.domain");
Session session = sessionFactory.openSession("http://username:password@localhost:7474");
```

If you don't want to or can't supply credentials as described above, the OGM can use the System properties `username` and `password` and supply them with each request to the Neo4j database.

Setting System properties

```
System.setProperty("username", user);
System.setProperty("password", pass);

SessionFactory sessionFactory = new SessionFactory("org.neo4j.example.domain");
Session session = sessionFactory.openSession("http://localhost:7474");
```

Compiler configuration

There is no explicit compiler configuration required for OGM 1.x

Session Configuration

In order to interact with mapped entities and the Neo4j graph, your application will require a `Session`, which is provided by the `SessionFactory`.

SessionFactory

The `SessionFactory` is needed by OGM to create instances of `org.neo4j.ogm.session.Session` as required. This also sets up the object-graph mapping metadata when constructed, which is then used across all `Session` objects that it creates. The packages to scan for domain object metadata should be provided to the `SessionFactory` constructor. Multiple packages may be provided as well. The SessionFactory must also be configured. There are two ways this can be done. Please see the section below on Configuration for further details.

Multiple packages

```
SessionFactory sessionFactory = new SessionFactory("first.package.domain", "second.package.domain",...);
```

Note that the `SessionFactory` should typically be set up once during life of your application.

Session

A `Session` is used to drive the object-graph mapping framework. It keeps track of the changes that have been made to entities and their relationships. The reason it does this is so that only entities and relationships that have changed get persisted on save, which is particularly efficient when working with large graphs. Note, however, that the `Session` **doesn't ever return cached objects** so there's no risk of getting stale data on load; it always hits the database.

The lifetime of the `Session` can be managed in code. For example, associated with single *fetch-update-save* cycle or unit of work.

If your application relies on long-running sessions and doesn't reload entities then you may not see changes made from other users and find yourself working with outdated objects. On the other hand,

if your sessions have too narrow a scope then your save operations can be unnecessarily expensive, as updates will be made to all objects if the session isn't aware of the those that were originally loaded.

There's therefore a trade off between the two approaches. In general, the scope of a `Session` should correspond to a "unit of work" in your application.

If you make sure you load fresh data at the beginning of each unit of work then data integrity shouldn't be a problem.

Programming model

This chapter covers the fundamentals of the programming model behind Neo4j OGM. It discusses the mapping mode, the annotations provided by the OGM and how to use them.

Under the bonnet

Metadata collection

Metadata is collected about persistent entities in `org.neo4j.ogm.metadata.Metadata` which provides it to any part of the library. This information is gathered by reading the class files directly rather than loading via reflection, resulting in much faster startup times.

The metadata holds all the required object-graph mapping information for each type. This metadata is discovered at start-up by specifying a list of packages in which all classes are scanned, including those in sub-packages. In order to omit a class from being metadata-mapped you should annotate it with `@org.neo4j.ogm.annotation.Transient`.

The Session object

`org.neo4j.ogm.session.Session` is a key component of the framework. The Session provides methods to load, save or delete object graphs from the database and also provides transaction support.

Explicit save

The OGM doesn't automatically commit when a transaction closes, so an explicit call to `save(…)` is required in order to persist changes to the database.

Fine-grained control via depth specification

Neo4j OGM includes the concept of persistence horizon (depth). On any individual request, the persistence horizon indicates how many relationships should be traversed in the graph when loading or saving data. A horizon of zero means that only the root object's properties will be loaded or saved, a horizon of 1 will include the root object and all its immediate neighbours, and so on. This attribute is enabled via a `depth` argument available on all session methods, but the OGM chooses sensible defaults so that you don't have to specify the depth attribute unless you want change the default values.

Default depth for loading

By default, loading an instance will map that object's simple properties and its immediately-related objects (i.e. depth = 1). This helps to avoid accidentally loading the entire graph into memory, but allows a single request to fetch not only the object of immediate interest, but also its closest neighbours, which are likely also to be of interest. This strategy attempts to strike a balance between loading too much of the graph into memory and having to make repeated requests for data.

If parts of your graph structure are deep and not broad (for example a linked-list), you can increase the load horizon for those nodes accordingly. Finally, if your graph will fit into memory, and you'd like to load it all in one go, you can set the depth to -1.

On the other hand when fetching structures which are potentially very "bushy" (e.g. lists of things that themselves have many relationships), you may want to set the load horizon to 0 (depth = 0) to avoid loading thousands of objects most of which you won't actually inspect.

Default depth for persisting

When persisting changes to the model, the default depth is -1. This means that **all affected** objects in the entity model that are reachable from the root object being persisted will be modified in the graph. This is the recommended approach because it means you can persist all your changes in one request. The OGM is able to detect which objects and relationships require changing, so you won't flood Neo4j with a bunch of objects that don't require modification. You can change the persistence depth to any value, but you should not make it less than the value used to load the corresponding data or you run the risk of not having changes you expect to be made actually being persisted in the graph.

In the example below, `session.save(user,1)` will persist all modified objects reachable from `user` up to one level deep. This includes `posts` and `groups` but not entities related to them, namely `author`, `comments`, `members` or `location`. A persistence depth of 0 i.e. `session.save(user,0)` will save only the properties on the user, ignoring any related entities. In this case, `fullName` is persisted but not friends, posts or groups.

Persistence Depth

```
public class User  {

   private Long id;
   private String fullName;
   private List<Post> posts;
   private List<Group> groups;

}

public class Post {

   private Long id;
   private String name;
   private String content;
   private User author;
   private List<Comment> comments;

}

public class Group {

   private Long id;
   private String name;
   private List<User> members;
   private Location location;

}
```

Entity Type Representation

For `@NodeEntity` classes, the simple names of the class and each of its parent classes (excluding `java.lang.Object`) is written as a node label. This node label is used in Cypher queries generated by the OGM to find objects of a particular type, and by labelling using superclasses as well it becomes possible to retrieve collections of entities as abstract super types.

Example domain model and labels

```
@NodeEntity
public abstract class DomainObject {
   @GraphId
   protected Long id;
}

public class Person extends DomainObject {
   ...
}

public class Lady extends Person {
   ...
}

public class Gentleman extends Person {
   ...
}

// creates a node with labels Gentleman:Person
session.save(new Gentleman());

// retrieve all ladies and gentlemen
Collection<Person> people = session.loadAll(Person.class);
```

The label applied to a node in the database can be configured by setting the value of the `label` property in the `@NodeEntity` annotation.

In the current version of the OGM, you must configure the relationship type to use by setting the `type` property in the `@RelationshipEntity` annotation as well as its corresponding `@Relationship` annotations.

Simplified Object-Graph Mapping

Neo4j OGM supports mapping annotated and non-annotated objects models. It's possible to save any POJO without annotations to the graph, as the framework applies conventions to decide what to do. This is useful in cases when you don't have control over the classes that you want to persist. The recommended approach, however, is to use annotations wherever possible, since this gives greater control and means that code can be refactored safely without risking breaking changes to the labels and relationships in your graph.

Annotated and non-annoted objects can be used within the same project without issue. There is an `EntityAccessStrategy` used to control how objects are read from or written to. The default implementation of this uses the following convention:

1. Annotated method (getter/setter)
2. Annotated field
3. Plain method (getter/setter)
4. Plain field

The object graph mapping comes into play whenever an entity is constructed from a node or relationship. This could be done explicitly during the lookup or create operations of the `Session` but also implicitly while executing any graph operation that returns nodes or relationships and expecting mapped entities to be returned.

Entities handled by the OGM must have one empty public constructor to allow the library to construct the objects.

Unless annotations are used to specify otherwise, the framework will attempt to map any of an object's "simple" fields to node properties and any rich composite objects to related nodes. A "simple" field is any primitive, boxed primitive or String or arrays thereof, essentially anything that naturally fits

into a Neo4j node property. For related entities the type of a relationship is inferred by the bean property name, as outlined in the examples below.

Defining node entities

Node entities are declared using the `@org.neo4j.ogm.annotation.NodeEntity` annotation. Relationship entities use the `@org.neo4j.ogm.annotation.RelationshipEntity` annotation.

@NodeEntity: The basic building block

The `@NodeEntity` annotation is used to declare that a POJO class is an entity backed by a node in the graph database. Entities handled by the OGM must have one empty public constructor to allow the library to construct the objects.

Fields on the entity are by default mapped to properties of the node. Fields referencing other node entities (or collections thereof) are linked with relationships.

`@NodeEntity` annotations are inherited from super-types and interfaces. It is not necessary to annotate your domain objects at every inheritance level.

If the `label` attribute is set then this will replace the default label applied to the node in the database. The default label is just the simple class name of the annotated entity. All parent classes are also added as labels so that retrieving a collection of nodes via a parent type is supported.

Entity fields can be annotated with `@Property`, `@GraphId`, `@Transient` or `@Relationship`. All annotations live in the `org.neo4j.ogm.annotation` package. Marking a field with the transient modifier has the same effect as annotating it with `@Transient`; it won't be persisted to the graph database.

Persisting an annotated entity

```
@NodeEntity
public class Actor extends DomainObject {

   @GraphId
   private Long id;

   @Property(name="name")
   private String fullName;

   @Relationship(type="ACTED_IN", direction=Relationship.OUTGOING)
   private List<Movie> filmography;

}

@NodeEntity(label="Film")
public class Movie {

   @GraphId Long id;

   @Property(name="title")
   private String name;

}
```

Saving a simple object graph containing one actor and one film using the above annotated objects would result in the following being persisted in Neo4j.

```
(:Actor:DomainObject {name:'Tom Cruise'})-[:ACTED_IN]->(:Film {title:'Mission Impossible'})
```

When annotating your objects, you can apply the annotations to either the fields or their accessor methods, but bear in mind the aforementioned `EntityAccessStrategy` ordering when annotating your domain model.

Persisting a non-annotated entity

```
public class Actor extends DomainObject {

   private Long id;
   private String fullName;
   private List<Movie> filmography;

}

public class Movie {

   private Long id;
   private String name;

}
```

In this case, a graph similar to the following would be persisted.

```
(:Actor:DomainObject {fullName:'Tom Cruise'})-[:FILMOGRAPHY]->(:Movie {name:'Mission Impossible'})
```

While this will map successfully to the database, it's important to understand that the names of the properties and relationship types are tightly coupled to the class's member names. Renaming any of these fields will cause parts of the graph to map incorrectly, hence the recommendation to use annotations.

@GraphId: Neo4j id field

This is a required field which must be of type `java.lang.Long`. It is used by Neo4j OGM to store the node or relationship-id to re-connect the entity to the graph. As such, user code should *never* assign a value to it.

It must not be a primitive type because then an object in a transient state cannot be represented, as the default value 0 would point to the reference node.

If the field is simply named 'id' then it is not necessary to annotate it with `@GraphId` as the OGM will use it automatically.

Entity Equality

Entity equality can be a grey area. There are many debatable issues, such as whether natural keys or database identifiers best describe equality and the effects of versioning over time. Neo4j OGM does not impose a dependency upon a particular style of `equals()` or `hashCode()` implementation. The graph-id field is directly checked to see if two entities represent the same node and a 64-bit hash code is used for dirty checking, so you're not forced to write your code in a certain way!

However, we do think it's important to mention that if you use the `@GraphId` field in your `hashCode()` method then this comes with a caveat. When you first persist an entity, its hashcode changes because the OGM populates the database ID on save.

That causes problems if you had inserted the newly created entity into a hash-based collection before saving. While that can be worked around, we strongly advise you adopt a convention of not relying upon the graph ID for object equality.

@Property: Optional annotation for property fields

As we touched on earlier, it is not necessary to annotate property fields as they are persisted by default. Fields that are annotated as `@Transient` or with `transient` are exempted from persistence. All fields that contain primitive values are persisted directly to the graph. All fields convertible to a `String`

using the conversion services will be stored as a string. Neo4j OGM includes default type converters that deal with the following types:

- `java.util.Date` to a String in the ISO 8601 format: "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
- `java.math.BigInteger` to a String property
- `java.math.BigDecimal` to a String property
- binary data (as byte[] or Byte[]) to base-64 String
- `java.lang.Enum` types using the enum's `name()` method and `Enum.valueOf()`

Collections of primitive or convertible values are stored as well. They are converted to arrays of their type or strings respectively. Custom converters are also specified by using `@Convert` - this is discussed in detail later on.

Node property names can be explicitly assigned by setting the `name` attribute. For example `@Property(name="last_name") String lastName`. The node property name defaults to the field name when not specified.

Property fields to be persisted to the graph must not be declared `final`.

Relating node entities

Since relationships are first-class citizens in Neo4j, associations between node entities are represented by relationships. In general, relationships are categorised by a type, and start and end nodes (which imply the direction of the relationship). Relationships can have an arbitrary number of properties. Neo4j OGM has special support to represent Neo4j relationships as entities too, but it is often not needed.

@Relationship: Connecting node entities

Every field of an entity that references one or more other node entities is backed by relationships in the graph. These relationships are managed by the OGM automatically.

The simplest kind of relationship is a single object reference pointing to another entity (1:1). In this case, the reference does not have to be annotated at all, although the annotation may be used to control the direction and type of the relationship. When setting the reference, a relationship is created when the entity is persisted. If the field is set to `null`, the relationship is removed.

Single relationship field

```
@NodeEntity
public class Movie {
    ...
    private Actor topActor;
}
```

It is also possible to have fields that reference a set of entities (1:N). Neo4j OGM supports the following types of entity collections-

- `java.util.Vector`
- `java.util.List`, backed by a `java.util.ArrayList`
- `java.util.SortedSet`, backed by a `java.util.TreeSet`
- `java.util.Set`, backed by a `java.util.HashSet`
- Arrays

Node entity with relationships

```
@NodeEntity
public class Actor {
    ...
    @Relationship(type = "TOP_ACTOR", direction = Relationship.INCOMING)
    private Set<Movie> topActorIn;

    @Relationship(type = "ACTS_IN")
    private Set<Movie> movies;
}
```

For graph to object mapping, the automatic transitive loading of related entities depends on the depth of the horizon specified on the call to `Session.load()`. The default depth of 1 implies that *related* node or relationship entities will be loaded and have their properties set, but none of their related entities will be populated.

If this `Set` of related entities is modified, the changes are reflected in the graph once the root object (`Actor`, in this case) is saved. Relationships are added, removed or updated according to the differences between the root object that was loaded and the corresponding one that was saved..

Neo4j OGM ensures by default that there is only one relationship of a given type between any two given entities. The exception to this rule is when a relationship is specified as either `OUTGOING` or `INCOMING` between two entities of the same type. In this case, it is possible to have two relationships of the given type between the two entities, one relationship in either direction.

If you don't care about the direction then you can specify `direction=Relationship.UNDIRECTED` which will guarantee that the path between two node entities is navigable from either side.

For example, consider the `PARTNER` relationship between two companies, where `(A)-[:PARTNER_OF] (B)` implies `(B)-[:PARTNER_OF] (A)`. The direction of the relationship does not matter; only the fact that a `PARTNER_OF` relationship exists between these two companies is of importance. Hence an `UNDIRECTED` relationship is the correct choice, ensuring that there is only one relationship of this type between two partners and navigating between them from either entity is possible.

The direction attribute on a `@Relationship` defaults to `OUTGOING`. Any fields or methods backed by an `INCOMING` relationship must be explicitly annotated with an `INCOMING` direction.

Using more than one Relationship of the same Type

In some cases, you want to model two different aspects of a conceptual relationship using the same relationship type. Here is a canonical example:

Clashing Relationship Type

```
@NodeEntity
class Person {
    private Long id;
    @Relationship(type="OWNS")
    private Car car;

    @Relationship(type="OWNS")
    private Pet pet;
...
}
```

This will work just fine, however, please be aware that this is only because the end node types (Car and Pet) are different types. If you wanted a person to own two cars, for example, then you'd have to use a `Collection` of cars or use differently-named relationship types.

Ambiguity in relationships

In cases where the relationship mappings could be ambiguous, the recommendation is that

- the objects be navigable in both directions and
- the @Relationship annotations are explicit. This means if the entity has setter methods, they must be annotated

Examples of ambiguous relationship mappings are multiple relationship types that resolve to the same types of entities, in a given direction, but whose domain objects are not navigable in both directions.

@RelationshipEntity: Rich relationships

To access the full data model of graph relationships, POJOs can also be annotated with `@RelationshipEntity`, making them relationship entities. Just as node entities represent nodes in the graph, relationship entities represent relationships. Such POJOs allow you to access and manage properties on the underlying relationships in the graph.

Fields in relationship entities are similar to node entities, in that they're persisted as properties on the relationship. For accessing the two endpoints of the relationship, two special annotations are available: `@StartNode` and `@EndNode`. A field annotated with one of these annotations will provide access to the corresponding endpoint, depending on the chosen annotation.

For controlling the relationship-type a `String` attribute called `type` is available on the `@RelationshipEntity` annotation. Like the simple strategy for labelling node entities, if this is not provided then the name of the class is used to derive the relationship type. As of the current version of the OGM, the `type` **must** be specified on the `@RelationshipEntity`.

You must include `@RelationshipEntity` plus exactly one `@StartNode` field and one `@EndNode` field on your relationship entity classes or the OGM will throw a MappingException when reading or writing. It is not possible to use relationship entities in a non-annotated domain model.

A simple Relationship entity

```
@NodeEntity
public class Actor {
    Long id;
    private Role playedIn;
}

@RelationshipEntity(type="PLAYED_IN")
public class Role {
    @GraphId   private Long relationshipId;
    @Property  private String title;
    @StartNode private Actor actor;
    @EndNode   private Movie movie;
}

@NodeEntity
public class Movie {
    private Long id;
    private String title;
}
```

Note that the `Actor` also contains a reference to a `Role`. This is important for persistence, **even when saving the `Role` directly**, because paths in the graph are written starting with nodes first and then relationships are created between them. Therefore, you need to structure your domain models so that relationship entities are reachable from node entities for this to work correctly.

Additionally, the OGM will not persist a relationship entity that doesn't have any properties defined. If

you don't want to include properties in your relationship entity then you should use a plain `@Relationship` instead. Multiple relationship entities which have the same property values and relate the same nodes are indistinguishable from each other and are represented as a single relationship by the OGM.

The `@RelationshipEntity` annotation must appear on all leaf subclasses if they are part of a class hierarchy representing relationship entities. This annotation is optional on superclasses.

Session

The `Session` provides the core functionality to persist objects to the graph and load them in a variety of ways.

Basic operations

Basic operations are limited to CRUD operations on entities and executing arbitrary Cypher queries; more low-level manipulation of the graph database is not possible.

There is no way to manipulate relationship- and node-objects directly.

Given that the Neo4j OGM framework is driven by Cypher queries alone, there's no way to work directly with `Node` and `Relationship` objects in remote server mode. Similarly, Traversal Framework operations are not supported, again because the underlying query-driven model doesn't handle it in an efficient way.

If you find yourself in trouble because of the omission of these features, then your best options are:

1. Write a Cypher query to perform the operations on the nodes/relationships instead
2. Write a Neo4j server extension and call it over REST from your application

Of course, there are pros and cons to both of these approaches, but these are largely outside the scope of this document. In general, for low-level, very high-performance operations like complex graph traversals you'll get the best performance by writing a server-side extension. For most purposes, though, Cypher will be performant and expressive enough to perform the operations that you need.

Entity-Persistence

`Session` allows to `save`, `load`, `loadAll` and `delete` entities with transaction handling and exception translation managed for you. The eagerness with which objects are retrieved is controlled by specifying the 'depth' argument to any of the load methods.

Cypher Queries

The `Session` also allows execution of arbitrary Cypher queries via its `query`, `queryForObject` and `queryForObjects` methods. Cypher queries that return tabular results should be passed into the `query` method. An `org.neo4j.ogm.model.Result` is returned. This consists of `org.neo4j.ogm.model.QueryStatistics` representing statistics of modifying cypher statements if applicable, and an `Iterable<Map<String,Object>>` containing the raw data, which can be either used as-is or converted into a richer type if needed. The keys in each `Map` correspond to the names listed in the return clause of the executed Cypher query.

In the current version, custom queries do not support paging, sorting or a custom depth. In addition, it does not support mapping a path to domain entities, as such, a path should not be returned from a Cypher query. Instead, return nodes and relationships to have them mapped to domain entities.

Modifications made to the graph via Cypher queries directly will not be reflected in your domain objects within the session.

Conversion

The object-graph mapping framework provides support for default and bespoke type conversions, which allow you to configure how certain data types are mapped to nodes or relationships in Neo4j.

Built-In Type Conversions

Neo4j OGM will automatically perform the following type conversions:

- `java.util.Date` to a String in the ISO 8601 format: "yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
- `java.math.BigInteger` to a String property
- `java.math.BigDecimal` to a String property
- binary data (as byte[] or Byte[]) to base-64 String as Cypher does not support byte arrays
- `java.lang.Enum` types using the enum's `name()` method and `Enum.valueOf()`

Two Date converters are provided "out of the box"

1. @DateString
2. @DateLong

By default, the OGM will use the @DateString converter as described above. However if you want to use a different date format, you can annotate your entity attribute accordingly:

Example of user-defined date format

```
public class MyEntity {

    @DateString("yy-MM-dd")
    private Date entityDate;
}
```

Alternatively, if you want to store Dates as long values, use the @DateLong annotation:

Example of date stored as a long value

```
public class MyEntity {

    @DateLong
    private Date entityDate;
}
```

Collections of primitive or convertible values are also automatically mapped by converting them to arrays of their type or strings respectively.

Custom Type Conversion

In order to define bespoke type conversions for particular members, you can annotate a field or method with `@Convert` to specify an implementation of

`org.neo4j.ogm.typeconversion.AttributeConverter` to use.

Example of custom type converter

```
public class MoneyConverter implements AttributeConverter<DecimalCurrencyAmount, Integer> {

   @Override
   public Integer toGraphProperty(DecimalCurrencyAmount value) {
       return value.getFullUnits() * 100 + value.getSubUnits();
   }

   @Override
   public DecimalCurrencyAmount toEntityAttribute(Integer value) {
       return new DecimalCurrencyAmount(value / 100, value % 100);
   }

}
```

You could then apply this to your class as follows:

```
@NodeEntity
public class Invoice {

   @Convert(MoneyConverter.class)
   private DecimalCurrencyAmount value;
   ...
}
```

Transactions

Neo4j is a transactional database, only allowing operations to be performed within transaction boundaries.

Transactions can be managed explicitly by calling the `beginTransaction()` method on the `Session` followed by a `commit()` or `rollback()` as required.

Transaction management

```
Transaction tx = session.beginTransaction();
Person person = session.load(Person.class,personId);
Concert concert= session.load(Concert.class,concertId);
Hotel hotel = session.load(Hotel.class,hotelId);

try {
    buyConcertTicket(person,concert);
    bookHotel(person, hotel);
    tx.commit();
}
catch (SoldOutException e) {
    tx.rollback();
}
tx.close();
```

In the example above, the transaction is committed only when both a concert ticket and hotel room is available, otherwise, neither booking is made.

If you do not manage a transaction in this manner, auto commit transactions are provided implicitly for `Session` methods such as `save`, `load`, `delete`, `execute` and so on.

Entity Management

Persisting Entities

Entity persistence is performed through the `save()` method on the underlying `Session` object.

Under the bonnet, the implementation of `Session` has access to the `MappingContext` that keeps track of the data that has been loaded from Neo4j during the lifetime of the session. Upon invocation of `save()` with an entity, it checks the given object graph for changes compared with the data that was loaded from the database. The differences are used to construct a Cypher query that persists the deltas to Neo4j before repopulating it's state based on the response from the database server.

Example 1. Persisting entities

```
@NodeEntity
public class Person {
   private String name;
   public Person(String name) {
      this.name = name;
   }
}

// Store Michael in the database.
Person p = new Person("Michael");
session.save(p);
```

Save Depth

As mentioned previously, `save(entity)` is overloaded as `save(entity, depth)`, where depth dictates the number of related entities to save starting from the given entity. The default depth, -1, will persist properties of the specified entity as well as every modified entity in the object graph reachable from it. A depth of 0 will persist only the properties of the specified entity to the database.

Specifying the save depth is handy when it comes to dealing with complex collections, that could potentially be very expensive to load.

If you're using this overloaded method, it's **strongly** recommended to use depth consistently between load and save invocations. If you don't then you may unexpectedly see relationships deleted or updates not persisting as you expect.

Example 2. Relationship save cascading

```
@NodeEntity
class Movie {
    String title;
    Actor topActor;
    public void setTopActor(Actor actor) {
        topActor = actor;
    }
}

@NodeEntity
class Actor {
    String name;
}

Movie movie = new Movie("Polar Express");
Actor actor = new Actor("Tom Hanks");

movie.setTopActor(actor);
```

Neither the actor nor the movie has been assigned a node in the graph. If we were to call

`session.save(movie)`, then the OGM would first create a node for the movie. It would then note that there is a relationship to an actor, so it would save the actor in a cascading fashion. Once the actor has been persisted, it will create the relationship from the movie to the actor. All of this will be done atomically in one transaction.

The important thing to note here is that if `session.save(actor)` is called instead, then only the actor will be persisted. The reason for this is that the actor entity knows nothing about the movie entity - it is the movie entity that has the reference to the actor. Also note that this behaviour is not dependent on any configured relationship direction on the annotations. It is a matter of Java references and is not related to the data model in the database.

In the following example, the actor and the movie are both managed entities, having both been previously persisted to the graph:

Example 3. Cascade for modified fields

```
actor.setBirthyear(1956);
session.save(movie);
```

In this case, even though the movie has a reference to the actor, the property change on the actor will not be persisted by the call to `save(movie)`. The reason for this is, as mentioned above, that cascading will only be done for fields that have been modified. Since the `movie.topActor` field has not been modified, it will not cascade the persist operation to the actor.

Sorting and Paging

Neo4j OGM supports Sorting and Paging of results when using the Session object. The Session object methods take independent arguments for Sorting and Pagination

Paging

```
Iterable<World> worlds = session.loadAll(World.class,
                                         new Pagination(pageNumber,itemsPerPage), depth)
```

Sorting

```
Iterable<World> worlds = session.loadAll(World.class,
                                         new SortOrder().add("name"), depth)
```

Sort in descending order

```
Iterable<World> worlds = session.loadAll(World.class,
                                         new SortOrder().add(SortOrder.Direction.DESC,"name"))
```

Sorting with paging

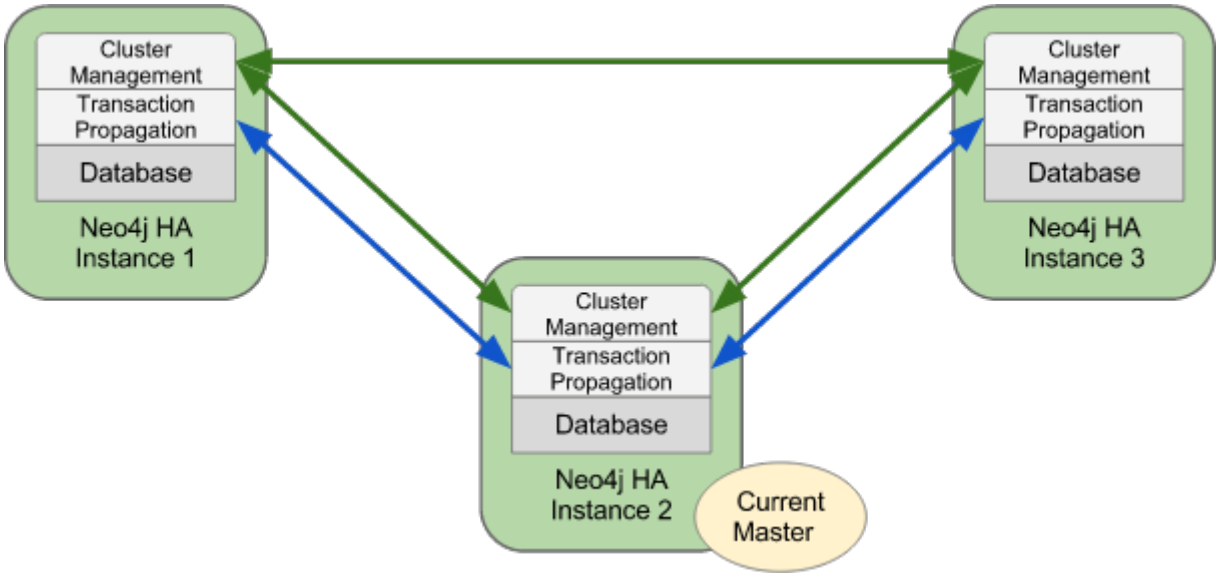
```
Iterable<World> worlds = session.loadAll(World.class,
                                         new SortOrder().add("name"), new Pagination(pageNumber
,itemsPerPage))
```

Neo4j OGM does not yet support sorting and paging on custom queries.

Configuring the OGM in an HA environment

Transaction Binding in HA mode

A typical Neo4j HA cluster will consist of a master node and a couple of slave nodes for providing failover capability and optionally for handling reads. (Although it is possible to write to slaves, this is uncommon because it requires additional effort to synchronise a slave with the master node)



When operating in HA mode, Neo4j does not make open transactions available across all nodes in the cluster. This means we must bind every request within a specific transaction to the same node in the cluster, or the commit will fail with `404 Not Found`.

Read-only Transactions

As of version 1.1.1 the OGM does not distinguish between WRITE transactions and READ-ONLY transactions. We cannot therefore bind read-only transactions to slaves and write transactions to master. A future version will address this deficiency, but in the meantime the only way to ensure that everything works as expected is to direct every transaction to master. There are a couple of ways to to achieve this.

Static binding to a designated master

Example cluster:

1. master: 192.168.0.55
2. slave1: 192.168.0.56
3. slave2: 192.168.0.67

OGM Binding to master ip address

```
Components.driver().setURI("http://192.168.0.55:7474");
```

We don't really recommend this approach, except for testing purposes and non-critical deployments. Firstly, it will only work if you always bring up the designated master first, and secondly, if the master goes down all subsequent transactions will fail until it is restarted. In HA mode, the cluster is able to elect a new master when this happens, but as of the current version of the OGM, there is no mechanism for querying the cluster to identify the current master. The solution in this case is to use a load balancer such as HAProxy that can do this for us. This is described in the next section.

Dynamic binding via a load balancer

In the Neo4j HA architecture, a cluster is typically fronted by a load balancer. The following example shows how to configure your application and set up HAProxy as a load balancer to route all requests to whichever machine in the cluster is currently identified as the master. Since only one machine can ever be the elected master, this should work exactly as we would like. Furthermore, should the elected master fail, a new server will be elected from the cluster as master and HAProxy will automatically route transactions to this server.

Example cluster fronted by HAProxy

1. haproxy: 10.0.2.200
2. neo4j-server1: 10.0.1.10
3. neo4j-server2: 10.0.1.11
4. neo4j-server3: 10.0.1.12

OGM Binding via HAProxy

```
    Components.driver().setURI("http://10.0.2.200");
```

Sample haproxy.cfg

```
global
    daemon
    maxconn 256

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http-in
    bind *:80
    default_backend neo4j

backend neo4j
    option httpchk GET /db/manage/server/ha/master
    server s1 10.0.1.10:7474 maxconn 32
    server s2 10.0.1.11:7474 maxconn 32
    server s3 10.0.1.12:7474 maxconn 32

listen admin
    bind *:8080
    stats enable
```

License

Creative Commons 3.0

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

Attribution

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike

If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions

You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See http://creativecommons.org/licenses/by-sa/3.0/ for further details. The full license text is available at http://creativecommons.org/licenses/by-sa/3.0/legalcode.