



Neo4j OGM - An Object Graph Mapping Library for Neo4j v3.1

Table of Contents

Introduction.....	2
What's new in OGM 3 ?	2
Requirements	2
Additional Resources	3
What is a graph database?	3
What is an OGM?	5
Tutorial.....	6
Introduction	6
Building the domain model	6
Configuring the OGM.....	8
Annotating the domain model.....	8
Interacting with the model	12
Conclusion	15
Reference	16
Introduction	16
Getting Started	17
Configuration	20
Annotating Entities	23
Indexing.....	33
Connecting to the Graph	34
Using the OGM Session.....	36
Transactions	41
Type Conversion.....	42
Filters	44
Events	45
Testing	49
High Availability (HA) Support	50
Appendix A: Migration from 2.1 to 3.0/3.1	55
Field access only.....	55
Configuration	55
Performance and unlimited load depth	55
Migration checklist.....	56
Appendix B: Design considerations.....	57
Variable-depth persistence.....	57
Smart object-mapping.....	57
User-definable Session lifetime	57
Appendix C: Frequently Asked Questions (FAQ)	58

© 2017 Neo4j, Inc.

License: [Creative Commons 4.0](#)

This is the Neo4j object-graph mapping (OGM) manual, authored by the Neo4j team.

The three parts of the manual are:

- [Introduction](#) — Introducing graph database concepts, Neo4j and object-graph mapping.
- [Tutorial](#) — Follow along as you get started using Neo4j OGM.
- [Reference](#) — Reference documentation for Neo4j OGM.

But before starting, let's see the most important new features.

Introduction

This chapter is an introduction to graph databases, Neo4j, and the Neo4j object-graph mapping library (OGM). It also outlines requirements and where to get support.

If you are already familiar with Neo4j and OGM, feel free to jump directly to the [tutorial](#) or [reference](#) sections.

What's new in OGM 3 ?

At a high level, here is what has changed in this third major version of OGM :

- Simplified and less intrusive development model

Technical database ids were previously required in the domain model ; this is no more the case. Ids can now be of any simple type, and generated using pluggable strategies. To avoid confusion about where to place annotations, they are now only valid on class attributes. See [Entity identifier](#) and [Migration from 2.1 to 3.0/3.1](#) for more details.

- Dynamic properties

This long awaited feature allows you to map node properties to dynamic structures (java maps). See [Dynamic properties](#).

- Improved performance

Retrieving data is more performant in OGM 3, particularly for deep depths, because querying leverages annotation information expressed in the domain model. This allows to build smarter queries, and only fetch useful mappable data.

- More flexible configuration

OGM 3 offers rich configuration possibilities detailed [here](#).

- New baselines

The minimum required versions are Java 8 and Neo4j 3.1.

- Better integration with various execution environments

Internally, OGM 3 relies on [FastClasspathScanner](#) (<https://github.com/lukehutch/fast-classpath-scanner>) which allows more reliable entity bytecode scanning, as well as better support application servers (like JBoss).



Check the [migration checklist](#) when upgrading from OGM 2.1

Requirements

Neo4j OGM 3.1.x at minimum, requires:

- JDK Version 8 and above.
- Neo4j Database 3.1.x / 3.2.x and above.

Additional Resources

Project metadata

- Version control - <https://github.com/neo4j/neo4j-ogm>
- Bugtracker - <https://github.com/neo4j/neo4j-ogm/issues>
- Release repository - <https://m2.neo4j.org/content/repositories/releases>
- Snapshot repository - <https://m2.neo4j.org/content/repositories/snapshots>

Getting Help or providing feedback

If you encounter issues or you are just looking for advice, feel free to use one of the links below:

- Talk and share with the community on the [SDN/OGM Slack channel](https://neo4j-users.slack.com) (<https://neo4j-users.slack.com>)
- The [sample project: OGM University](https://github.com/neo4j-examples/neo4j-ogm-university) (<https://github.com/neo4j-examples/neo4j-ogm-university>). This project is used in the [Tutorial](#).
- The [sample project: Movies](https://github.com/neo4j-examples/movies-java-spring-data-neo4j) (<https://github.com/neo4j-examples/movies-java-spring-data-neo4j>).
- For more detailed questions, use [Neo4j OGM on StackOverflow](http://stackoverflow.com/questions/tagged/neo4j-ogm) (<http://stackoverflow.com/questions/tagged/neo4j-ogm>)
- Use the [templates](https://github.com/neo4j-examples/neo4j-sdn-ogm-issue-report-template) (<https://github.com/neo4j-examples/neo4j-sdn-ogm-issue-report-template>) for reporting issues (they can also be used to bootstrap your projects).
- For professional support feel free to contact Neo Technology or GraphAware.

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.

Introducing Neo4j

[Neo4j](http://neo4j.com/) (<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/developer-manual/current/introduction/#graphdb-neo4j>).
- 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](https://neo4j.com/docs/#api) (<https://neo4j.com/docs/#api>).

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 with Cypher

Neo4j provides a graph query language called [Cypher](http://neo4j.com/docs/stable/cypher-query-lang.html) (<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.

Here are some examples of how easy Cypher is to use (These queries work with the "Movies" data set that comes installed with Neo4j browser)

Names and birthplaces of Actors who appeared in a Matrix movie

```
MATCH (movie:Movie)-[:ACTS_IN]-(actor)
WHERE movie.title =~ 'Matrix.*'
RETURN actor.name, actor.birthplace
```

All movie titles the user "michal" rated more than 3 stars

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

User michal's friends who rated a movie more than 3 stars

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

Learning more

The jumping off ground for learning about Neo4j is neo4j.com (<https://neo4j.com/>). Here is a list of other useful resources:

- The [Neo4j documentation](https://neo4j.com/docs/) (<https://neo4j.com/docs/>) introduces Neo4j and contains links to getting started guides, reference documentation and tutorials.
- The [online sandbox](https://neo4j.com/sandbox/) (<https://neo4j.com/sandbox/>) provides a convenient way to interact with a Neo4j instance in combination with the online [tutorial](https://neo4j.com/developer/get-started/). (<https://neo4j.com/developer/get-started/>)

- Neo4j [Java Bolt Driver](https://neo4j.com/docs/developer-manual/3.1/drivers/) (<https://neo4j.com/docs/developer-manual/3.1/drivers/>).
- Several [books](https://neo4j.com/books/) (<https://neo4j.com/books/>) available for purchase and [videos](https://www.youtube.com/neo4j) (<https://www.youtube.com/neo4j>) to watch.

What is an OGM?

An OGM (Object Graph Mapper) maps nodes and relationships in the graph to objects and references in a domain model. Object instances are mapped to nodes while object references are mapped using relationships, or serialized to properties (e.g. references to a Date). JVM primitives are mapped to node or relationship properties. An OGM abstracts the database and provides a convenient way to persist your domain model in the graph and query it without using low level drivers. It also provides the flexibility to the developer to supply custom queries where the queries generated by the OGM are insufficient.

Introducing the Neo4j OGM

Developing Java Business Applications often requires mapping rich domain models to your database. The Neo4j-OGM library is a pure Java library that can persist (annotated) domain objects using Neo4j. It uses Cypher statements to handle those operations in Neo4j.

The OGM supports tracking changes to minimize necessary updates and transitive persistence (reading and updating neighborhoods of an object).

The connection to Neo4j handled by a driver layer, which can use the binary protocol, HTTP or Neo4j's embedded APIs.

Tutorial

This chapter is a tutorial that takes the reader through steps necessary to get started with the Neo4j OGM.

Introduction

Neo4j OGM University is a demo application for the Neo4j OGM library that allows you to manage the Departments, Teaching Staff, Subjects, Students and Classes of a fictitious educational institution: Hilly Fields Technical College.

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

- Groovy
- Ratpack
- Neo4j OGM
- AngularJS
- Bootstrap

The application's architecture involves a RESTful server interfacing with a rich single page application that is designed to show off the performance and capabilities of the OGM.

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

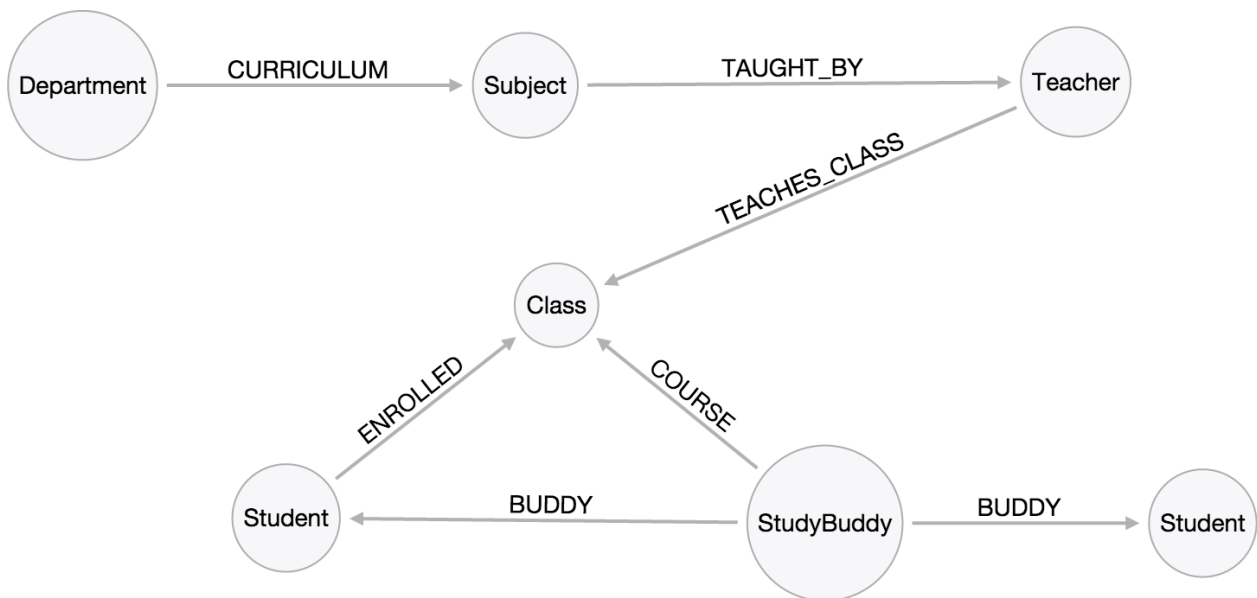
Building the domain model

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.



When we translate this model to Groovy, it ends up being pretty straightforward:

```

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

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

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

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

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

class Enrollment {
    Student student;
    Course course;
    Date enrolledDate;
}

```

When a student enrolls for a course, we're also going to keep track of the enrollment date.

In the model, 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**.

The important thing to take away here is that the OGM supports Neo4j's philosophy of whiteboard friendly domain models. By focusing on the model the code almost writes itself.

Configuring the OGM

Neo4j OGM supports several drivers:

- Bolt - the lightning fast native driver for Neo4j.
- HTTP - the original transactional HTTP endpoint for remote Neo4j deployments.
- Embedded - for embedded deployments within a Java application.

Our sample application will use the Bolt driver.

Setting up with Gradle

The demo application uses [Gradle](https://gradle.org/) (<https://gradle.org/>) as a build system.

Before we can use the library, we need to add a dependency.

Gradle dependencies for Neo4j OGM

```
compile "org.neo4j:neo4j-ogm-core:3.1.0"
runtime "org.neo4j:neo4j-ogm-bolt-driver:3.1.0"
```

Refer to [Dependency Management](#) for more information on dependencies.

Connecting to the database

We configure the database parameters by using the configuration builder.

```
Configuration configuration = new Configuration.Builder()
    .uri("bolt://localhost")
    .credentials("neo4j", "password")
    .build();

SessionFactory sessionFactory = new SessionFactory(configuration, "com.mycompany.app.domainclasses");
```

Annotating the domain model

Much like Hibernate or JPA, the OGM allows you to annotate your POJOs in order to map them to nodes, relationships and properties in the graph.

Node Entities

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 in the code we wrote earlier.

Note that we're overriding the default label for a `Course` with `Class`

```

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

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

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

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

@NodeEntity
class Student {
    String name;
    Set<Enrollment> enrollments;
    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. Again, we are going to modify the code we saw in the last section:

```

@NodeEntity
class Department {
    String name;

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

@NodeEntity
class Subject {
    String name;

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

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

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

@NodeEntity
class Teacher {
    String name;

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

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

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

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

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

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

@NodeEntity
class Student {
    String name;

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

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

```

Relationship Entities

Sometimes something isn't quite a Node entity.

In this demo the only remaining class to annotate is `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")
class Enrollment {

    @StartNode
    Student student;

    @EndNode
    Course course;

    Date enrolledDate;

}

```

Identifiers

Every node and relationship persisted to the graph must have an id. The OGM uses this to identify and re-connect the entity to the graph in memory. Identifier may be either a primary id or a native graph id.

- primary id - any property annotated with `@Id`, set by the user and optionally with `@GeneratedValue` annotation
- native id - this id corresponds to the id generated by the Neo4j database when a node or relationship is first saved, must be of type `Long`



Do not rely on native id for long running applications. Neo4j will reuse deleted node id's. It is recommended users come up with their own unique identifier for their domain objects (or use a UUID).

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.

If you plan on implementing `hashCode` and `equals` make sure it **does not** make use of the native id. See [Node Entities](#) for more information.

```

abstract class Entity {

    @Id @GeneratedValue
    private Long id;

    public Long getId() {
        return id;
    }

}

```

Our entities will now extend this class, for example

```

@NodeEntity
class Department extends Entity {
    String name;

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

    Department() {

    }

}

```

No Arg Constructor

We are almost there!

The OGM also requires a public no-args constructor to be able to construct objects from all our annotated entities. We'll make sure all our entities have one.

Converters

Neo4j supports `Numeric`, `String`, `boolean` and arrays of these as property values.

How do we handle the `enrolledDate` since `Date` is not a valid data type?

Luckily for us, 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 its `Long` representation and back is handled by the OGM when persisting and loading from the graph.

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

    Long id;

    @StartNode
    Student student;

    @EndNode
    Course course;

    @DateLong
    Date enrolledDate;

    Enrollment() {
    }
}
```

Interacting with the model

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

Sessions

The smart object mapping capability is provided by the `Session` object. 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 Configuration = ... // provide configuration as seen before
    private final static SessionFactory sessionFactory = new SessionFactory(configuration,
    "school.domain");
    private static Neo4jSessionFactory factory = new Neo4jSessionFactory();

    public static Neo4jSessionFactory getInstance() {
        return factory;
    }

    // prevent external instantiation
    private Neo4jSessionFactory() {
    }

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

The `SessionFactory` constructor accepts packages that are to be scanned for annotated domain entities.

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.



We use here the `SessionFactory` with the package of domain classes as a parameter. This sets up an in-memory embedded database. In your application, you would also pass the configuration to connect to your actual database.

The `Session` keeps track of changes made to entities and relationships and persists ones that have been modified on save. Once an entity is tracked by the session, reloading this entity within the scope of the same session will result in the session cache returning the previously loaded entity. However, the subgraph in the session will expand if the entity or its related entities retrieve additional relationships from the graph.

For the purpose of this demo application, we'll use short living sessions - a new session per web request - to avoid stale data issues.

Our university application will use the following operations:


```
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`. Let's write a `GenericService` to deal with common `Session` operations.

```
abstract class GenericService<T> implements Service<T> {  
    private static final int DEPTH_LIST = 0  
    private static final int DEPTH_ENTITY = 1  
    protected Session session = Neo4jSessionFactory.getInstance().getNeo4jSession()  
  
    @Override  
    Iterable<T> findAll() {  
        return session.loadAll(getEntityType(), DEPTH_LIST)  
    }  
  
    @Override  
    T find(Long id) {  
        return session.load(getEntityType(), id, DEPTH_ENTITY)  
    }  
  
    @Override  
    void delete(Long id) {  
        session.delete(session.load(getEntityType(), id))  
    }  
  
    @Override  
    T createOrUpdate(T entity) {  
        session.save(entity, DEPTH_ENTITY)  
        return find(entity.id)  
    }  
  
    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.

**Hilly Fields Technical College**
Timendi Causa Est Nesciri

Student Registration and Curriculum Management

Home Reports Entities




Class 27:Biopolymer Physics

Back Save Delete


Field	Value
Name	Biopolymer Physics
Subject	Physics
Teacher	Mr Marker
Students Enrolled	<input checked="" type="checkbox"/> Isabella Santana

Loading relationships is not required 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.

**Hilly Fields Technical College**
Timendi Causa Est Nesciri

Student Registration and Curriculum Management

Home Reports Entities



Departments

Create a new Department

Find

0:Engineering

1:Science

2:Mathematics

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`.

```
class StudyBuddyServiceImpl extends GenericService<StudyBuddy> implements StudyBuddyService {  
    @Override  
    Iterable<StudyBuddy> findAll() {  
        return session.loadAll(StudyBuddy, 1)  
    }  
  
    @Override  
    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  
    Class<StudyBuddy> getEntityType() {  
        return StudyBuddy.class  
    }  
}
```

The `query` provided by the `Session` can return a domain object, a collection of them, or a special wrapped object called a `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) (<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

This chapter is the reference documentation for Neo4j OGM. It covers the programming model, APIs, concepts, annotations and technical details of the Neo4j OGM.

Introduction

Neo4j OGM is a fast object-graph mapping library for Neo4j, optimised for server-based installations utilising Cypher.

It aims to simplify development with the Neo4j graph database and like JPA, it uses annotations on simple POJO domain objects to do so.

With a focus on performance, the OGM 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.

Overview

This reference documentation is broken down into sections to help the user understand specifics of how the OGM works.

Getting started

Getting started can sometimes be a chore. What versions of the OGM do you need? Where do you get them from? What build tool should you use? [Getting Started](#) is the perfect place to well... get started!

Configuration

Drivers, logging, properties, configuration via Java. How to make sense of all the options? [Configuration](#) has got you covered.

Annotating your Domain Objects

To get started with your OGM application, you need only your domain model and 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. Because Neo4j is a schema-free database, the OGM uses a simple mechanism to map Java types to Neo4j nodes using labels. Relationships between entities are first class citizens in a graph database and therefore worth a [section of it's own](#) describing their usage in Neo4j OGM.

Connecting to the Database

Managing how you connect to the database is important. [Connecting to the Database](#) has all the details on what needs to happen to get you up and running.

Indexing and Primary Constraints

Indexing is an important part of any database. The Neo4j OGM provides a variety of features to

support the management of Indexes as well as the ability to query your domain objects by something other than the internal Neo4j id. [Indexing](#) has everything you will want to know when it comes to getting that working.

Interacting with the Graph Model

Neo4j OGM offers a [session](#) for interacting with the mapped entities and the Neo4j graph database. Neo4j uses transactions to guarantee the integrity of your data and Neo4j OGM supports this fully. The implications of this are described in the transactions section. 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 in the introduction chapter.

Type Conversion

The OGM 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. See [Type Conversion](#) for more details.

Filtering your domain objects

Filters provides a simple API to append criteria to your stock `Session.loadX()` behaviour. This is covered in more detail in [Filters](#).

Reacting to Persistence events

The Events mechanism allows users to register event listeners for handling persistence events related both to top-level objects being saved as well as connected objects. [Event handling](#) discusses all the aspects of working with events.

Testing in your application

Sometimes you want to be able to run your tests against an in-memory version of the OGM. [Testing](#) goes into more detail of how to set that up.

Support for High Availability

For those using Neo4j Enterprise, support for high availability is extremely important. The chapter on [High Availability](#) goes into all the options the OGM provides to support this.

Getting Started

Versions

Consult the version table to determine which version of the OGM to use with a particular version of Neo4j and related technologies.

Compatibility

Neo4j OGM Version	Neo4j Version	Bolt Version [#]	Spring Data Neo4j Version	Spring Boot Version
3.1.0+	3.1.x, 3.2.x, 3.3.x	1.5.0+ (compatible with 1.4.0+)	5.1.0+ (compatible with 5.0.0+)	2.0.0+
3.0.0+	3.1.x, 3.2.x, 3.3.x	1.4.0+	5.0.0+	2.0.0+
2.1.0+	2.3.x, 3.0.x, 3.1.x	1.1.0+	4.2.0+	1.5.0+
2.0.2+	2.3.x, 3.0.x	1.0.0+	4.1.2 - 4.1.6+	1.4.x
2.0.1*	2.2.x, 2.3.x	1.0.0-RC1	4.1.0 - 4.1.1	1.4.x
1.1.5*	2.1.x, 2.2.x, 2.3.x	N/A	4.0.0+	1.4.x

* These versions are no longer actively developed or supported.

Not applicable to Embedded and HTTP drivers

Transitive dependencies

The following table list transitive dependencies between specific versions of projects related to OGM. When reporting issues or asking for help on StackOverflow or neo4j-users slack channel always verify versions used (e.g through `mvn dependency:tree`) and report them as well.

Spring Boot Version	Spring Data Neo4j Version	Neo4j OGM Version	Bolt Version
2.0.0	5.1.0	3.1.0	1.5.0
2.0.0	5.0.0 (default for Spring Boot 2.0)	3.0.0	1.4.3
1.5.7	4.2.7	2.1.3	1.2.3
1.4.6	4.1.7	2.0.5	1.0.6



These versions can be overridden manually in `pom.xml` or `build.gradle` files.

Dependency Management

For building an application, your build automation tool needs to be configured to include the Neo4j OGM dependencies.

The OGM dependencies consist of `neo4j-ogm-core`, together with the relevant dependency declarations on the driver you want to use. OGM provides support for connecting to Neo4j by configuring one of the following Drivers:

- `neo4j-ogm-http-driver` - Uses HTTP to communicate between the OGM and a remote Neo4j instance.
- `neo4j-ogm-embedded-driver` - Connects directly to the Neo4j database engine.
- `neo4j-ogm-bolt-driver` - Uses native Bolt protocol to communicate between the OGM and a remote Neo4j instance.

If you're not using a particular driver, you don't need to declare it.

Neo4j OGM projects can be built using Maven, Gradle or any other build system that utilises Maven's artifact repository structure.

Maven

In the `<dependencies>` section of your `pom.xml` add the following:

Maven dependencies

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-ogm-core</artifactId>
  <version>3.1.0</version>
  <scope>compile</scope>
</dependency>

<!-- Only add if you're using the HTTP driver -->
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-ogm-http-driver</artifactId>
  <version>3.1.0</version>
  <scope>runtime</scope>
</dependency>

<!-- Only add if you're using the Embedded driver -->
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-ogm-embedded-driver</artifactId>
  <version>3.1.0</version>
  <scope>runtime</scope>
</dependency>

<!-- Only add if you're using the Bolt driver -->
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-ogm-bolt-driver</artifactId>
  <version>3.1.0</version>
  <scope>runtime</scope>
</dependency>
```

If you plan on using a development (i.e. **SNAPSHOT**) version of the OGM you will need to add the following to the `<repositories>` section of your `pom.xml`:

Neo4j Snapshot Repository

```
<repository>
  <id>neo4j-snapshot-repository</id>
  <name>Neo4j Maven 2 snapshot repository</name>
  <url>http://m2.neo4j.org/content/repositories/snapshots</url>
</repository>
```

Gradle

Ensure the following dependencies are added to your `build.gradle`:

Gradle dependencies

```
dependencies {
  compile 'org.neo4j:neo4j-ogm-core:3.1.0'
  runtime 'org.neo4j:neo4j-ogm-http-driver:3.1.0' // Only add if you're using the HTTP driver
  runtime 'org.neo4j:neo4j-ogm-embedded-driver:3.1.0' // Only add if you're using the Embedded driver
  runtime 'org.neo4j:neo4j-ogm-bolt-driver:3.1.0' // Only add if you're using the Bolt driver
}
```

If you plan on using a development (i.e. **SNAPSHOT**) version of the OGM you will need to add the following section of your `build.gradle`:

Neo4j Snapshot Repository

```
repositories {
  maven { url "http://m2.neo4j.org/content/repositories/snapshots" }
}
```

Configuration

Configuration method

There are several ways to supply configuration to the OGM:

- using a properties file
- programmatically using Java
- by providing an already configured Neo4j Java driver instance

These methods are described below. They are also available as code in the [examples](#).

Using a properties file

Properties file on classpath:

```
ConfigurationSource props = new ClasspathConfigurationSource("my.properties");  
Configuration configuration = new Configuration.Builder(props).build();
```

Properties file on filesystem:

```
ConfigurationSource props = new FileConfigurationSource("/etc/my.properties");  
Configuration configuration = new Configuration.Builder(props).build();
```

Programmatically using Java

In cases where you are not be able to provide configuration via a properties file you can configure the OGM programmatically instead.

The `Configuration` object provides a fluent API to set various configuration options. This object then needs to be supplied to the `SessionFactory` constructor in order to be configured.

By providing a Neo4j driver instance

Just configure the driver as you would do for direct access to the database, and pass the driver instance to the session factory.

This method allows the greatest flexibility and gives you access to the full range of low level configuration options.

Example providing a bolt driver instance to OGM

```
org.neo4j.driver.v1.Driver nativeDriver = ...;  
Driver ogmDriver = new BoltDriver(nativeDriver);  
new SessionFactory(ogmDriver, ...);
```

Driver Configuration

For configuration through properties file or configuration builder the driver is automatically inferred from given URI. Empty URI means embedded driver with impermanent database.

HTTP Driver

Table 1. Basic HTTP Driver Configuration

ogm.properties	Java Configuration
[source, properties] ---- URI=http://user:password@localhost:7474 ----	[source, java] ---- Configuration configuration = new Configuration.Builder() .uri("http://user:password@localhost:7474") .build() ----

Bolt Driver

Note that for the [URI](#), if no port is specified, the default Bolt port of [7687](#) is used. Otherwise, a port can be specified with [bolt://neo4j:password@localhost:1234](#).

Also, the bolt driver allows you to define a connection pool size, which refers to the maximum number of sessions per URL. This property is optional and defaults to [50](#).

Table 2. Basic Bolt Driver Configuration

ogm.properties	Java Configuration
[source, properties] ---- URI=bolt://neo4j:password@localhost connection.pool.size=150 ----	[source, java] ---- Configuration configuration = new Configuration.Builder() .uri("bolt://neo4j:password@localhost") .setConnectionPoolSize(150) .build() ----

A timeout to the database with the Bolt driver can be set by updating your Database's [neo4j.conf](#). The exact setting to change can be [found here](#) (http://neo4j.com/docs/operations-manual/current/reference/configuration-settings/#config_dbms.transaction.timeout).

Embedded Driver

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.



As of 2.1.0 the Neo4j OGM embedded driver no longer ships with the Neo4j kernel. Users are expected to provide this dependency through their dependency management system. See [Getting Started](#) for more details.

Table 3. Permanent Data Store Embedded Driver Configuration

ogm.properties	Java Configuration
[source, properties] ---- URI=file:///var/tmp/neo4j.db ----	[source, java] ---- Configuration configuration = new Configuration.Builder() .uri("file:///var/tmp/neo4j.db") .build() ----

To use an impermanent data store which will be deleted on shutdown of the JVM, you just omit the URI attribute.

Table 4. Impermanent Data Store Embedded Driver Configuration

ogm.properties	Java Configuration
[source, properties] ---- # Leave empty ----	[source, java] ---- Configuration configuration = new Configuration.Builder().build() ----

Configuration in an Unmanaged Extension

When your application is running as unmanaged extension inside the Neo4j server itself, you will need to set up OGM configuration slightly differently. Neo4j provides [PluginLifecycle](#) SPI that allows to

initialize extensions. Extend `OgmPluginInitializer` and list the full class name in `META-INF/services/org.neo4j.server.plugins.PluginLifecycle`:

```
public class MyApplicationPluginInitializer extends OgmPluginInitializer {  
    public MyApplicationPluginInitializer() {  
        super(MyDomain.class.getPackage().getName());  
    }  
}
```

This provides `SessionFactory` as injectable in your resources:

```
@Path("/movies")  
public static class MovieService {  
    @Context  
    private SessionFactory sessionFactory;  
    ...  
}
```



Don't forget to list your resources in `dbms.unmanaged_extension_classes` property in Neo4j configuration file as you would with any other unmanaged extension.

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	Java Configuration
[source, properties] ---- # embedded URI=http://user:password@localhost:7474 # separately username="user" password="password" ----	[source, java] ---- Configuration configuration = new Configuration.Builder() .uri("bolt://user:password@localhost") .build() Configuration configuration = new Configuration.Builder() .credentials("user", "password") .build() ----

Note: Currently only Basic Authentication is supported by the OGM. If you need to use more advanced authentication scheme, use the native driver configuration method.

Transport Layer Security (TLS/SSL)

The Bolt and HTTP drivers also allow you to connect to Neo4j over a secure channel. These rely on Transport Layer Security (aka TLS/SSL) and require the installation of a signed certificate on the server.

In certain situations (e.g. some cloud environments) it may not be possible to install a signed certificate even though you still want to use an encrypted connection.

To support this, both drivers have configuration settings allowing you to bypass certificate checking, although they differ in their implementation.



Both of these strategies leave you vulnerable to a MITM attack. You should probably not use them unless your servers are behind a secure firewall.

Bolt

ogm.properties	Java Configuration
[source, properties] ---- #Encryption level (TLS), optional, defaults to REQUIRED. #Valid values are NONE,REQUIRED encryption.level=REQUIRED #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 ----	[source, java] ---- Configuration config = new Configuration.Builder()encryptionLevel("REQUIRED") .trustStrategy("TRUST_ON_FIRST_USE") .trustCertFile("/tmp/cert") .build(); ----

TRUST_ON_FIRST_USE means that the Bolt Driver will trust the first connection to a host to be safe and intentional. On subsequent connections, the driver will verify that the host is the same as on that first connection.

HTTP

ogm.properties	Java Configuration
[source, properties] ---- trust.strategy = ACCEPT_UNSIGNED ----	[source, java] ---- Configuration configuration = new Configuration.Builder() .trustStrategy("ACCEPT_UNSIGNED") .build() ----

The **ACCEPT_UNSIGNED** strategy permits the HTTP Driver to accept Neo4j's default **snakeoil.cert** (and any other) unsigned certificate when connecting over HTTPS.

Bolt connection testing

In order to prevent some network problems while accessing a remote database, you may want to tell the Bolt driver to test connections from the connection pool.

This is particularly useful when there are firewalls between the application tier and the database.

You can do that with the connection liveness parameter which indicates the interval at which the connections will be tested. A value of 0 indicates that the connection will always be tested. A negative value indicates that the connection will never be tested.

ogm.properties	Java Configuration
[source, properties] ---- # interval, in milliseconds, to check for stale db connections (test-on-borrow) connection.liveness.check.timeout=1000 ----	[source, java] ---- Configuration config = new Configuration.Builder()connectionLivenessCheckTimeout(1000) .build(); ----

Eager connection verification

OGM by default does not connect to Neo4j server on application startup. This allows you to start the application and database independently and Neo4j will be accessed on first read/write. To change this behaviour set the property **verify.connection** (or **Builder.verifyConnection(boolean)**) to true. This settings is valid only for Bolt and HTTP drivers.

Logging

Neo4j OGM uses SLF4J to log statements. In production, you can set the log level in a file called **logback.xml** to be found at the root of the classpath. Please see the [Logback manual](http://logback.qos.ch/manual/) (<http://logback.qos.ch/manual/>) for further details.

Annotating Entities

@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 (excluding `java.lang.Object`) are also added as labels so that retrieving a collection of nodes via a parent type is supported.

Entity fields can be annotated with annotations like `@Property`, `@Id`, `@GeneratedValue`, `@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 {

    @Id @GeneratedValue
    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 {

    @Id @GeneratedValue 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 choose to NOT apply the annotations on the fields. OGM will then use conventions to determine property names in the database for each field.

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.

Please read [Non-annotated properties and best practices](#) for more details and best practices on this.

@Properties: dynamically mapping properties to graph

A `@Properties` annotation tells OGM to map values of a Map field in a node or relationship entity to properties of a node or a relationship in the graph.

The property names are derived from field name or `prefix`, `delimiter` and keys in the Map. For example Map field with name `address` containing following entries:

```
"street" => "Downing Street"  
"number" => 10
```

will map to following node/relationship properties

```
address.street=Downing Street  
address.number=10
```

Supported types for keys in the Map are String and Enum.

The values in the Map can be of any Java type equivalent to Cypher types. If full type information is provided other Java types are also supported.

If annotation parameter `allowCast` is set to true then types that can be cast to corresponding Cypher types are allowed as well.



The original type cannot be deduced and the value will be deserialized to corresponding type - e.g. when Integer instance is put to `Map<String, Object>` it will be deserialized as Long.

```
@NodeEntity
public class Student {

    @Properties
    private Map<String, Integer> properties = new HashMap<>();

    @Properties
    private Map<String, Object> properties = new HashMap<>();

}
```

Runtime managed labels

As stated above, the label applied to a node is the contents of the `@NodeEntity` label property, or if not specified, it will default to the simple class name of the entity. Sometimes it might be necessary to add and remove additional labels to a node at *runtime*. We can do this using the `@Labels` annotation. Let's provide a facility for adding additional labels to the `Student` entity:

```
@NodeEntity
public class Student {

    @Labels
    private List<String> labels = new ArrayList<>();

}
```

Now, upon save, the node's labels will correspond to the entity's class hierarchy *plus* whatever the contents of the backing field are. We can use one `@Labels` field per class hierarchy - it should be exposed or hidden from sub-classes as appropriate.

Runtime labels must not conflict with static labels defined on node entities.



In a typical situation OGM issues one request per node entity type when saving node entities to the database. Using many distinct labels will result into many requests to the database (one request per unique combination of labels).

@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.
- The `@Relationship` annotations are explicit.

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.

Ordering

Neo4j doesn't have any ordering on relationships, so the relationships are fetched without any specific ordering. If you want to impose order on collections of relationships you have several options:

- use a `SortedSet` and implement `Comparable`
- sort relationships in `@PostLoad` annotated method

You can sort either by a property of a related node or by relationship property. To sort by relationship property you need to use a relationship entity. See [@RelationshipEntity: Rich relationships](#).

@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, although it's converted into `SNAKE_CASE` to honour the naming conventions of Neo4j relationships. As of the current version of the OGM, the `type` **must** be specified on the `@RelationshipEntity` annotation as well as its corresponding `@Relationship` annotations.



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;
    @Relationship(type="PLAYED_IN") private Role playedIn;
}

@RelationshipEntity(type="PLAYED_IN")
public class Role {
    @Id @GeneratedValue 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.

A note on JSON serialization

Looking at the example given above the circular dependency on the class level between the node and the rich relationship can easily be spotted. It will not have any effect on your application as long as you do not serialize the objects. One kind of serialization that is used today is JSON serialization using the Jackson mapper. This mapper library will be used if data gets exported in frameworks like SpringBoot or JavaEE. Traversing the object tree it will hit the part when it visits a **Role** after visiting an **Actor**. Obvious it will then find the **Actor** object and visit this again, and so on. This will end up in a **StackOverflowError**. To break this parsing cycle it is mandatory to support the mapper by providing annotation to your class(es). This can be done by adding either **@JsonIgnore** on the property that causes the loop or **@JsonIgnoreProperties**.

Suppress infinite traversing

```
@NodeEntity
public class Actor {
    Long id;

    // needs knowledge about the attribute name in the relationship
    @JsonIgnoreProperty("actor")
    @Relationship(type="PLAYED_IN") private Role playedIn;
}

@RelationshipEntity(type="PLAYED_IN")
public class Role {
    @Id @GeneratedValue private Long relationshipId;
    @Property private String title;

    // direct way to suppress the serialization, but only makes sense if this is not the entry object.
    @JsonIgnore
    @StartNode private Actor actor;

    @EndNode private Movie movie;
}
```

Entity identifier

Every node and relationship persisted to the graph must have an ID. The OGM uses this to identify and re-connect the entity to the graph in memory. Identifier may be either a primary id or a native graph id (*the technical id attributed by Neo4j at node creation time*).

For primary id use the `@Id` on a field of any supported type or a field with provided `AttributeConverter`. A unique index is created for such property (if index creation is enabled). User code should either set the id manually when the entity instance is created or id generation strategy should be used. It is not possible to store an entity with null id value and no generation strategy.



Specifying primary id on a relationship entity is possible, but lookups by this id are slow, because Neo4j database doesn't support schema indexes on relationships.

For native graph id use `@Id @GeneratedValue` (with default strategy `InternalIdStrategy`). The field type must be `Long`. This id is assigned automatically upon saving the entity to the graph and 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.



Do not rely on this ID for long running applications. Neo4j will reuse deleted node ID's. It is recommended users come up with their own unique identifier for their domain objects (or use a UUID).

An entity can be looked up by this either type of id by using `Session.load(Class<T>, ID)` and `Session.loadAll(Class<T>, Collection<ID>)` methods.

It is possible to have both natural and native id in one entity. In such situation lookups prefer the primary id.

If the field of type `Long` is simply named 'id' then it is not necessary to annotate it with `@Id @GeneratedValue` as the OGM will use it automatically as native id.

@GraphId: Neo4j id field

The `@GraphId` annotation is superseded by `@Id @GeneratedValue` and exists for backwards compatibility. It is deprecated and will eventually be removed.



Do not rely on this ID for long running applications. Neo4j will reuse deleted node ID's. It is recommended users come up with their own unique identifier for their domain objects (or use a UUID).

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!



You are free to write your `equals` and `hashCode` in a domain specific way for managed entities. However, **we strongly advise developers to not use the `@GraphId` field in these implementations**. This is because when you first persist an entity, its hashcode changes because the OGM populates the database ID on save. This causes problems if you had inserted the newly created entity into a hash-based collection before saving.

Id Generation Strategy

If the `@Id` annotation is used on its own it is expected that the field will be set by the application code. To automatically generate and assign a value of the property the annotation `@GeneratedValue` can be used.

The `@GeneratedValue` annotation has optional parameter `strategy`, which can be used to provide a custom id generation strategy. The class must implement `org.neo4j.ogm.id.IdStrategy` interface. The strategy class can either supply no argument constructor - in which case OGM will create an instance of the strategy and call it. For situations where some external context is needed an externally created instance can be registered with SessionFactory by using `SessionFactory.register(IdStrategy)`.

Optimistic locking with @Version annotation

Optimistic locking is supported by OGM to provide concurrency control. To use optimistic locking define a field annotated with `@Version` annotation. The field is then managed by OGM and used to perform optimistic locking checks when updating entities. The type of the field must be `Long` and an entity may contain only one such field.

Typical scenario where optimistic locking is used then looks like follows:

- new object is created, version field contains `null` value
- when the object is saved the version field is set to 0 by OGM
- when a modified object is saved the version provided in the object is checked against a version in the database during the update, if successful then the version is incremented both in the object and in the database
- if another transaction modified the object in the meantime (and therefore incremented the version) then this is detected and an `OptimisticLockingException` is thrown

Optimistic locking check is performed for

- updating properties of nodes and relationship entities
- deleting nodes via `Session.delete(T)`
- deleting relationship entities via `Session.delete(T)`

- deleting relationship entities detected through `Session.save(T)`

When an optimistic locking failure happens following operations are performed on the Session:

- object which failed the optimistic locking check is removed from the context so it can be reloaded
- in case a default transaction is used it is rolled back
- in case a manual transaction is used then it is **not** rolled back, but because the update may contain multiple statements which are checked eagerly it is not defined what updates were actually performed in the database and it is advised to rollback the transaction. If you know you updates consists of single modification you may however choose to reload the object and continue the transaction.

@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.time.Instant` to a String in the ISO 8601 with timezone format: "yyyy-MM-dd'T'HH:mm:ss.SSSZ"
- `java.time.LocalDate` to a String in the ISO 8601 with format: "yyyy-MM-dd"
- `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`.

@PostLoad

A method annotated with `@PostLoad` will be called once the entity is loaded from the database.

Non-annotated properties and best practices

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.



The support for non-annotated domain classes might be dropped in the future, to allow startup optimizations.

Annotated and non-annotated objects can be used within the same project without issue.

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.

Indexing

Indexing is used in Neo4j to quickly find nodes and relationships from which to start graph operations.

Indexes and Constraints

Indexes based on labels and properties are supported with the `@Index` annotation. Any property field annotated with `@Index` will use have an appropriate schema index created. For `@Index(unique=true)` a constraint is created.

You may add as many indexes or constraints as you like to your class. If you annotate a field in a class that is part of an inheritance hierarchy then the index or constraint will only be added to that class's label.

Primary Constraints



The `primary` property of the `@Index` annotation is deprecated since OGM 3 and should not be used. The primary key is solely provided by the `@Id` annotation. See [Entity identifier](#) for more information.

Composite Indexes and Node Key Constraints

Composite indexes based on label and multiple properties are supported with `@CompositeIndex` annotation. The annotation is to be placed at the class level. All properties specified must exist within the class or one of its superclasses. It is possible to create multiple composite indexes by repeating the annotation.

Providing `unique=true` parameter will create a node key constraint instead of a composite index.



This feature is only supported by Neo4j Enterprise 3.2 and higher.

Existence constraints

Existence constraints for a property is supported with `@Required` annotation. It is possible to annotate properties in both node entities and relationship entities. For node entities the label of declaring class

is used to create the constraint. For relationship entities the relationship type is used - such type must be defined on leaf class.



This feature is only supported by Neo4j Enterprise 3.1 and higher.

Index Creation

By default index management is set to **None**.

If you would like the OGM to manage your schema creation there are several ways to go about it.

Only classes marked with **@Index**, **@CompositeIndex** or **@Required** will be used. Indexes will always be generated with the containing class's label and the annotated property's name. An abstract class containing indexes or constraints must have **@NodeEntity** annotation present. Index generation behaviour can be defined in **ogm.properties** by defining a property called: **indexes.auto** and providing a value of:

Below is a table of all options available for configuring Auto-Indexing.

Option	Description	Properties Example	Java Example
none (default)	Nothing is done with index and constraint annotations.	-	-
validate	Make sure the connected database has all indexes and constraints in place before starting up	<code>indexes.auto=validate</code>	<code>config.setAutoIndex("validate");</code>
assert	Drops all constraints and indexes on startup then builds indexes based on whatever is represented in OGM by @Index . Handy during development	<code>indexes.auto=assert</code>	<code>config.setAutoIndex("assert");</code>
update	Builds indexes based on whatever is represented in OGM by @Index . Indexes will be changed to constraints and vice versa if the definition in db differs from metadata. Handy during development	<code>indexes.auto=update</code>	<code>config.setAutoIndex("update");</code>
dump	Dumps the generated constraints and indexes to a file. Good for setting up environments. none: Default. Simply marks the field as using an index.	<code>indexes.auto=dump</code> <code>indexes.auto.dump.dir=<a directory></code> <code>indexes.auto.dump.filename=<a filename></code>	<code>config.setAutoIndex("dump");</code> <code>config.setDumpDir("XXX");</code> <code>config.setDumpFilename("XXX");</code>

Connecting to the Graph

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 **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.



The `SessionFactory` is an expensive object to create because it scans all the requested packages to build up metadata. It should typically be set up once during life of your application.

Create SessionFactory with `Configuration` instance

As seen in the configuration section, this is done by providing the `SessionFactory` a configuration object:

```
SessionFactory sessionFactory = new SessionFactory(configuration, "com.mycompany.app.domainclasses");
```

Create SessionFactory with `Driver` instance

This can be done by providing to the `SessionFactory` a driver instance:

```
SessionFactory sessionFactory = new SessionFactory(driver, "com.mycompany.app.domainclasses");
```

Embedded driver instance

If a pre-configured embedded database is needed, it can be passed into the embedded driver. It is possible to either use a configuration file

```
GraphDatabaseService db = new GraphDatabaseFactory()  
    .newEmbeddedDatabaseBuilder(new File(storeDir))  
    .loadPropertiesFromFile(pathToConfigFile)  
    .newDatabase();
```

or set the setting parameters programmatically.

```
GraphDatabaseService db = new GraphDatabaseFactory()  
    .newEmbeddedDatabaseBuilder(new File(storeDir))  
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )  
    .newDatabase();
```

and pass them into the `EmbeddedDriver`.

```
EmbeddedDriver driver = new EmbeddedDriver(db)  
  
SessionFactory sessionFactory = new SessionFactory(driver, "com.mycompany.app.domainclasses");
```

Multiple entity packages

Multiple packages may be provided as well. If you would rather just pass in specific classes you can also do that via an overloaded constructor.

Multiple packages

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

Using the OGM Session

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

Session Configuration

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. Once an entity is tracked by the session, reloading this entity within the scope of the same session will result in the session cache returning the previously loaded entity. However, the subgraph in the session will expand if the entity or its related entities retrieve additional relationships from the graph.

If you want to fetch fresh data from the graph, then this can be achieved by using a new session or clearing the current sessions context using `Session.clear()`.

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 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 a too narrow 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.

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.

Persisting entities

`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.

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 its state based on the response from the database server.

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.

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. 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. 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.

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 be** persisted by the call to `save(movie)`. The reason for this is, as mentioned above, that cascading will be done for fields that have been modified and reachable from the root object being saved.

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.


```
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;  
}
```

Loading Entities

Entities can be loaded from the OGM through the use of the `session.loadXXX()` methods or via `session.query()/session.queryForObject()` which will accept your own Cypher queries (See section below on [cypher queries](#)).

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.

Load depth

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.



When loading entities with a custom depth less than the one used previously to load the entity within the session, existing relationships will not be flushed from the session; only new entities and relationships are added. This means that reloading entities will always result in retaining related objects loaded at the highest depth within the session for those entities. If it is required to load entities with a lower depth than previously requested, this must be done on a new session, or after clearing your current session with `Session.clear()`.

Query Strategy

When OGM loads entities through `load*` methods (including ones with filters) it uses `LoadStrategy` to generate the `RETURN` part of the query.

Available load strategies are

- **schema load strategy** - uses metadata on domain entities and pattern comprehensions to retrieve nodes and relationships (default since OGM 3.0)
- **path load strategy** - uses paths from root node to fetch related nodes, `p=(n)-[0..]-()` (default before OGM 3.0)

The strategy can be overridden globally by calling `SessionFactory.setLoadStrategy(strategy)` or for single session only (e.g. when different strategy is more effective for given query) by calling `Session.setLoadStrategy(strategy)`

Cypher queries

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 plug in the HTTP Driver or the Bolt Driver when deploying your code into a production client-server environment.

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 which returns an `Result`. This consists of `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.

`queryForObject` specifically queries for entities and as such, queries supplied to this method must return nodes and not individual properties.

Query methods that retrieve mapped objects may be used in cases where the query generated by load strategy does not have sufficient performance.

Such queries should return nodes and optionally relationships. For a relationship to be mapped both start and end node must be returned.

Query methods returning particular domain type collect the result from all result columns and nested structures in these (e.g. collected lists, maps etc..) and return as single `Iterable<T>`. Use `Result.Session.query(java.lang.String, java.util.Map<java.lang.String, ?>)` to retrieve only objects in particular column.



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.

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.

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

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



make sure to always close the transaction by wrapping it in a `try-with-resources` block or by calling `close()` in a `finally` block.

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.

Transactions are by default `READ_WRITE` but can also be opened as `READ_ONLY`.

Opening a read only transaction

```
Transaction tx = session.beginTransaction(Transaction.Type.READ_ONLY);  
...
```

This is important for clustering where the type of transaction is used to route requests to servers. See [the high availability section](#).

Type 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.time.Instant` to a String in the ISO 8601 with timezone format: "yyyy-MM-dd'T'HH:mm:ss.SSSZ"
- `java.time.LocalDate` to a String in the ISO 8601 with format: "yyyy-MM-dd"
- Any object that extends `java.lang.Number` 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 `java.util.Date` or `java.time.Instant` as long values, use the `@DateLong` annotation:

Example of date stored as a long value

```
public class MyEntity {  
    @DateLong  
    private Date entityDate;  
}
```

`java.time.Instant` dates are stored in the database using UTC.

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



Collections are not supported for `java.time.Instant` and `java.time.LocalDate`.

Lenient conversion

It is possible to explicitly assign the build-in converter annotations to the corresponding fields. This provides the advantage of being able to use the `lenient` attribute that will get be read by the converters. The supported annotations are `@DateString`, `@EnumString` and `@NumberString`. Example of lenient converter usage

```
public class MyEntity {  
    @DateString(lenient = true)  
    private Date entityDate;  
}
```

The lenient feature is currently only supported by string-based converters to allow the conversion of blank strings from the database.

Custom Type Conversion

In order to define bespoke type conversions for particular members, you can annotate a field or method with `@Convert`. One of either two convert implementations can be used. For simple cases where a single property maps to a single field, with type conversion, specify an implementation of `AttributeConverter`.

Example of mapping a single property to a field

```
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;
    ...
}
```

When more than one node property is to be mapped to a single field, use:
`CompositeAttributeConverter`.

Example of mapping multiple node entity properties onto a single instance of a type

```
/**
 * This class maps latitude and longitude properties onto a Location type that encapsulates both of these
 * attributes.
 */
public class LocationConverter implements CompositeAttributeConverter<Location> {

    @Override
    public Map<String, ?> toGraphProperties(Location location) {
        Map<String, Double> properties = new HashMap<>();
        if (location != null) {
            properties.put("latitude", location.getLatitude());
            properties.put("longitude", location.getLongitude());
        }
        return properties;
    }

    @Override
    public Location toEntityAttribute(Map<String, ?> map) {
        Double latitude = (Double) map.get("latitude");
        Double longitude = (Double) map.get("longitude");
        if (latitude != null && longitude != null) {
            return new Location(latitude, longitude);
        }
        return null;
    }
}
```

And just as with an `AttributeConverter`, a `CompositeAttributeConverter` could be applied to your class as follows:

```
@NodeEntity
public class Person {

    @Convert(LocationConverter.class)
    private Location location;
    ...
}
```

Filters

Filters provide a mechanism for customising the where clause of Cypher generated by OGM. They can be chained together with boolean operators, and associated with a comparison operator. Additionally, each filter contains a `FilterFunction`. A filter function can be provided when the filter is instantiated, otherwise, by default a `PropertyComparison` is used.

In the example below, we're return a collection containing any satellites that are manned.

Example of using a Filter

```
Collection<Satellite> satellites = session.loadAll(Satellite.class, new Filter("manned", EQUALS, true));
```

Example of chained Filters

```
Filter mannedFilter = new Filter("manned", equals, true);
Filter landedFilter = new Filter("landed", equals, false);

Filters satelliteFilter = mannedFilter.and(landedFilter);
```



The filters should be considered as immutable. In previous versions, you could change filter values after instantiation, this is not the case anymore.

Events

Neo4j OGM supports persistence events. This section describes how to intercept update and delete events.

You may also check the `@PostLoad` annotation which is described [here](#).

Event types

There are four types of events:

```
Event.LIFECYCLE.PRE_SAVE
Event.LIFECYCLE.POST_SAVE
Event.LIFECYCLE.PRE_DELETE
Event.LIFECYCLE.POST_DELETE
```

Events are fired for every `@NodeEntity` or `@RelationshipEntity` object that is created, updated or deleted, or otherwise affected by a save or delete request. This includes:

- The top-level objects or objects being created, modified or deleted.
- Any connected objects that have been modified, created or deleted.
- Any objects affected by the creation, modification or removal of a relationship in the graph.



Events will only fire when one of the `session.save()` or `session.delete()` methods is invoked. Directly executing Cypher queries against the database using `session.query()` will not trigger any events.

Interfaces

The Events mechanism introduces two new interfaces, `Event` and `EventListener`.

The Event interface

The `Event` interface is implemented by `PersistenceEvent`. Whenever an application wishes to handle an event it will be given an instance of `Event`, which exposes the following methods:

```
public interface Event {
    Object getObject();
    LIFECYCLE getLifecycle();

    enum LIFECYCLE {
        PRE_SAVE, POST_SAVE, PRE_DELETE, POST_DELETE
    }
}
```

The Event Listener interface

The `EventListener` interface provides methods allowing implementing classes to handle each of the different `Event` types:

```
public interface EventListener {  
  
    void onPreSave(Event event);  
    void onPostSave(Event event);  
    void onPreDelete(Event event);  
    void onPostDelete(Event event);  
  
}
```



Although the `Event` interface allows you to retrieve the event type, in most cases, your code won't need it because the `EventListener` provides methods to capture each type of event explicitly.

Registering an EventListener

There are two way to register an event listener:

- on an individual `Session`
- across multiple sessions by using a `SessionFactory`

In this example we register an anonymous `EventListener` to inject a UUID onto new objects before they're saved

```
class AddUuidPreSaveEventListener implements EventListener {  
  
    void onPreSave(Event event) {  
        DomainEntity entity = (DomainEntity) event.getObject():  
        if (entity.getId() == null) {  
            entity.setUUID(UUID.randomUUID());  
        }  
    }  
    void onPostSave(Event event) {  
    }  
    void onPreDelete(Event event) {  
    }  
    void onPostDelete(Event event) {  
    }  
}  
  
EventListener eventListener = new AddUuidPreSaveEventListener();  
  
// register it on an individual session  
session.register(eventListener);  
  
// remove it.  
session.dispose(eventListener);  
  
// register it across multiple sessions  
sessionFactory.register(eventListener);  
  
// remove it.  
sessionFactory.deregister(eventListener);
```




It's possible and sometimes desirable to add several `EventListener` objects to the session, depending on the application's requirements. For example, our business logic might require us to add a UUID to a new object, as well as manage wider concerns such as ensuring that a particular persistence event won't leave our domain model in a logically inconsistent state. It's usually a good idea to separate these concerns into different objects with specific responsibilities, rather than having one single object try to do everything.

Using the EventListenerAdapter

The `EventListener` above is fine, but we've had to create three methods for events we don't intend to handle. It would be preferable if we didn't have to do this each time we needed an `EventListener`.

The `EventListenerAdapter` is an abstract class providing a no-op implementation of the `EventListener` interface. If you don't need to handle all the different types of persistence event you can create a subclass of `EventListenerAdapter` instead and override just the methods for the event types you're interested in.

For example:

```
class PreSaveEventListener extends EventListenerAdapter {
    @Override
    void onPreSave(Event event) {
        DomainEntity entity = (DomainEntity) event.getObject();
        if (entity.id == null) {
            entity.UUID = UUID.randomUUID();
        }
    }
}
```

Disposing of an EventListener

Something to bear in mind is that once an `EventListener` has been registered it will continue to respond to any and all persistence events. Sometimes you may want only to handle events for a short period of time, rather than for the duration of the entire session.

If you're done with an `EventListener` you can stop it from firing any more events by invoking `session.dispose(...)`, passing in the `EventListener` to be disposed of.



The process of collecting persistence events prior to dispatching them to any `EventListener`s adds a small performance overhead to the persistence layer. Consequently, the OGM is configured to suppress the event collection phase if there are no `EventListener`s registered with the Session. Using `dispose()` when you're finished with an `EventListener` is good practice!

To remove an event listener across multiple sessions use the `deregister` method on the `SessionFactory`.

Connected objects

As mentioned previously, events are not only fired for the top-level objects being saved but for all their connected objects as well.

Connected objects are any objects reachable in the domain model from the top-level object being saved. Connected objects can be many levels deep in the domain model graph.

In this way, the Events mechanism allows us to capture events for objects that we didn't explicitly save

ourselves.

```
// initialise the graph
Folder folder = new Folder("folder");
Document a = new Document("a");
Document b = new Document("b");
folder.addDocuments(a, b);

session.save(folder);

// change the names of both documents and save one of them
a.setName("A");
b.setName("B");

// because `b` is reachable from `a` (via the common shared folder) they will both be persisted,
// with PRE_SAVE and POST_SAVE events being fired for each of them
session.save(a);
```

Events and types

When we delete a Type, all the nodes with a label corresponding to that Type are deleted in the graph. The affected objects are not enumerated by the Events mechanism (they may not even be known). Instead, **_DELETE** events will be raised for the Type:

```
// 2 events will be fired when the type is deleted.
// - PRE_DELETE Document.class
// - POST_DELETE Document.class
session.delete(Document.class);
```

Events and collections

When saving or deleting a collection of objects, separate events are fired for each object in the collection, rather than for the collection itself.

```
Document a = new Document("a");
Document b = new Document("b");

// 4 events will be fired when the collection is saved.
// - PRE_SAVE a
// - PRE_SAVE b
// - POST_SAVE a
// - POST_SAVE b

session.save(Arrays.asList(a, b));
```

Event ordering

Events are partially ordered. **PRE_** events are guaranteed to fire before any **POST_** event within the same **save** or **delete** request. However, the **internal** ordering of the **PRE_** events and **POST_** events with the request is undefined.

Example: Partial ordering of events

```
Document a = new Document("a");
Document b = new Document("b");

// Although the save order of objects is implied by the request, the PRE_SAVE event for `b`
// may be fired before the PRE_SAVE event for `a`, and similarly for the POST_SAVE events.
// However, all PRE_SAVE events will be fired before any POST_SAVE event.

session.save(Arrays.asList(a, b));
```

Relationship events

The previous examples show how events fire when the underlying **node** representing an entity is updated or deleted in the graph. Events are also fired when a save or delete request results in the modification, addition or deletion of a **relationship** in the graph.

For example, if you delete a Document object that is a member of a Folder's documents collection, events will be fired for the Document as well as the Folder, to reflect the fact that the relationship between the folder and the document has been removed in the graph.

Example: Deleting a Document attached to a Folder

```
Folder folder = new Folder();
Document a = new Document("a");
folder.addDocuments(a);
session.save(folder);

// When we delete the document, the following events will be fired
// - PRE_DELETE a
// - POST_DELETE a
// - PRE_SAVE folder
// - POST_SAVE folder
session.delete(a);
```

Note that the **folder** events are **_SAVE** events, not **_DELETE** events. The **folder** was not deleted.



The event mechanism does not try to synchronise your domain model. In this example, the folder is still holding a reference to the Document, even though it no longer exists in the graph. As always, your code must take care of domain model synchronisation.

Event uniqueness

The event mechanism guarantees to not fire more than one event of the same type for an object in a save or delete request.

Example: Multiple changes, single event of each type

```
// Even though we're making changes to both the folder node, and its relationships,
// only one PRE_SAVE and one POST_SAVE event will be fired.
folder.removeDocument(a);
folder.setName("newFolder");
session.save(folder);
```

Testing

Doing integration testing with OGM requires a few basic steps :

- Add the **neo4j-ogm-test** artifact in you maven / gradle configuration
- Declare the **Neo4jRule** JUnit rule, to setup a Neo4j test server
- Setup the OGM configuration and **SessionFactory**

An example of a full running configuration can be found in the [issue templates](https://github.com/neo4j-examples/neo4j-sdn-ogm-issue-report-template/blob/master/ogm-3.0/src/test/java/org/neo4j/ogm/test/OgmTestCase.java) (<https://github.com/neo4j-examples/neo4j-sdn-ogm-issue-report-template/blob/master/ogm-3.0/src/test/java/org/neo4j/ogm/test/OgmTestCase.java>)

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>
```

High Availability (HA) Support



The clustering features are only available in Neo4j Enterprise Edition.

Neo4j offers two separate solutions for ensuring redundancy and performance in a high-demand production environment:

- Causal Clustering
- Highly Available (HA) Cluster

Neo4j 3.1 introduced Causal Clustering – a brand-new architecture using the state-of-the-art Raft protocol – that enables support for ultra-large clusters and a wider range of cluster topologies for data center and cloud.

A Neo4j HA cluster is comprised of a single master instance and zero or more slave instances. All instances in the cluster have full copies of the data in their local database files. The basic cluster configuration usually consists of three instances.

Causal Clustering

To find out more about Causal Clustering architecture please see [the reference](https://neo4j.com/docs/operations-manual/current/clustering/) (<https://neo4j.com/docs/operations-manual/current/clustering/>).

Causal Clustering only works with the Neo4j Bolt Driver (**1.1.0** onwards). Trying to set this up with the HTTP or Embedded Driver will not work. The Bolt driver will fully handle any load balancing, which operate in concert with the Causal Cluster to spread the workload. New cluster-aware sessions, managed on the client-side by the Bolt drivers, alleviate complex infrastructure concerns for developers.

Configuring the OGM

Not cluster specific side note: you may also want to configure [connection testing](#).

To use clustering, simply configure your Bolt URI to use the bolt routing protocol:

```
URI=bolt+routing://instance0
```

`instance0` must be one of your core cluster group (that accepts reads and writes).

Design considerations for clustering

In this section we go through important points to be aware of when using causal clustering.

- Review hardware and cluster configuration
- Target replica servers when possible
- Use bookmarks to read your own writes
- Plan for failure

Hardware and cluster configuration

Hardware, and particularly network, can have a great impact on cluster stability. The deployment scenario also plays a critical role. It has to be carefully chosen, each configuration having its strengths and weaknesses.

Please read carefully the [causal cluster reference](https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/) (<https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/>) to plan the best topology according to your needs.

You can also provide additional core instances in `URIS` property, separated by a comma. The `URI` property still needs to be set and will be tried first, followed by entries from `URIS` property. Same credentials are used for all instances. All listed instances must be core servers.

```
URI=bolt+routing://instance0
URIS=bolt+routing://instance1,bolt+routing://instance2
```

Target replica servers when possible

By default all `Session`'s `Transaction`s are set to read/write. This means reads and writes will always hit the core cluster. To offload the core servers and improve performance, it is advised if possible to route traffic to the replica servers. This is done in the application code, by declaring sessions / transactions as read-only. You can call `session.beginTransaction(Transaction.Type)` with `READ` to do that.



This is not always possible. You may only do this if you can afford to read some slightly outdated data.

Use bookmarks to read your own writes

Causal consistency allows you to specify guarantees around query ordering, including the **ability to read your own writes**, view the last data you read, and later on, committed writes from other users. The Bolt drivers collaborate with the core servers to ensure that all transactions are applied in the same order using a concept of a bookmark.

The cluster returns a bookmark when it commits an update transaction, so then the driver links a bookmark to the user's next transaction. The server that received query **starts this new bookmarked transaction only when its internal state reached the desired bookmark**. This ensures that the view of related data is always consistent, that all servers are eventually updated, and that users reading and re-reading data always see the same — and the latest — data.

If you have multiple application tier JVM instances you will need to manage this state across them. The `Session` object allows you to retrieve bookmarks through the use of `Session.getLastBookmark()` and start new transactions with given bookmark through `Session.beginTransaction(type, bookmarks)`.



Do not generalize the use of bookmarks as they have impact on latency.

Retry mechanisms

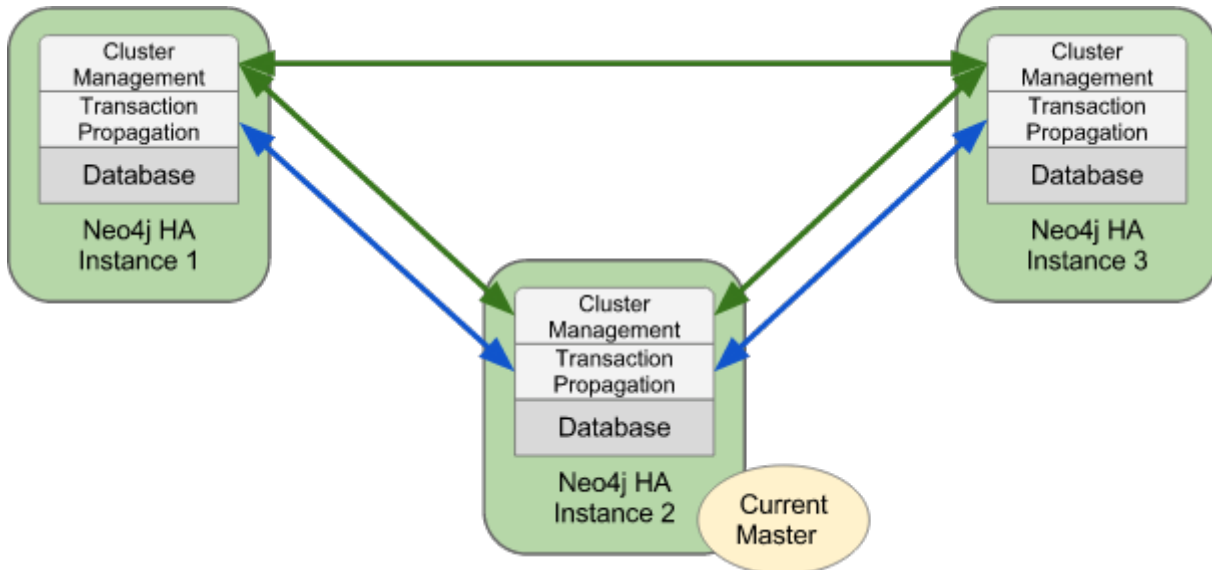
The driver does its best to ensure a stable communication between the application tier and the database. It handles low level failures (like connection loss), but cannot do much about higher level failures (like cluster unavailability). However, due to the nature of distributed platforms, failures arise. When the cluster is split among several datacenters, network issues can cause cluster instability. Cluster members not being able to talk to each other can make the cluster, for example, fall in read only mode, or trigger leader re-election.

For critical applications, these failures have to be anticipated, and also managed at the architecture or application level. Even if the driver handles some low level retries, it is not always enough in case of instability, as an application may involve complex business logic, and require coarse grained units of work.

Solutions like application retries or message queuing are good candidates to handle this kind of scenario.

Highly Available (HA) Cluster

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.)



Transaction binding in HA mode

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 2.0.5 read-only transactions are supported by the OGM.

Drivers

The Drivers have been updated to transmit additional information about the transaction type of the current transaction to the server.

- The HTTP Driver implementation sets a HTTP Header "X-WRITE" to "1" for READ_WRITE transactions (the default) or to "0" for READ_ONLY ones.
- The Embedded Driver can support both READ_ONLY and READ_WRITE (as of version [2.1.0](#)).
- The native Bolt Driver can support both READ_ONLY and READ_WRITE (as of version [2.1.0](#)).

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 write requests to whichever machine in the cluster is currently identified as the master, with read requests being distributed to any available machine in the cluster on a round-robin basis.

This configuration will also ensure that requests against a specific transaction are directed to the server where the transaction was created.

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

```
new Configuration.Builder().uri("http://10.0.2.200").build();
```

Sample haproxy.cfg

```
global
    daemon
    maxconn 256

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

frontend http-in
    bind *:80
    acl write_hdr hdr_val(X-WRITE) eq 1
    use_backend neo4j-master if write_hdr
    default_backend neo4j-cluster

backend neo4j-cluster
    balance roundrobin
    # create a sticky table so that requests with a transaction id are always sent to the correct server
    stick-table type integer size 1k expire 70s
    stick match path,word(4,/)
    stick store-response hdr(Location),word(6,/)
    option httpchk GET /db/manage/server/ha/available
    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

backend neo4j-master
    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
```


Appendix A: Migration from 2.1 to 3.0/3.1

This migration guide should help with migration from OGM 2.1.x version to 3.0.x or 3.1.x.

Field access only

Probably the most invasive change is how OGM accesses entity fields. Up to version 2.1.x any setters/getters took precedence before direct field access. This was also the reason why some annotations had to be on both setters and getters/fields and was a frequent source of confusion and many questions on Slack and StackOverflow.

Since version 3.0.0 all properties and relationships are accessed (read and write) by direct field access only. All annotations on getters or setters must be moved to field directly. If there is an annotation duplicated on both field and setter/getter, only the annotation on the field should stay.



The `@Target` type of the annotations (`@Relationship`, `@Property`, etc.) was changed to `ElementType.FIELD`, so you can easily identify places which needs updating by trying to compile your project.

It was possible to perform some initialisation, validation or logic in annotated setters. New annotation `@PostLoad` was introduced to allow for any such action after the entity is fully hydrated.

Configuration

New class `Configuration.Builder` was introduced for Java configuration using properties or property file. See [configuration](#) section and JavaDoc for details.

`Components` class was removed, if you need to provide custom driver instance use appropriate constructor of `SessionFactory`.

You no longer need to provide the driver class name as in previous versions. The driver is automatically inferred from given `URI`.

Performance and unlimited load depth

In order to improve loading performance, some optimizations are now used by OGM when querying. A new load policy based on [list comprehensions](https://neo4j.com/docs/developer-manual/current/cypher/syntax/lists/#cypher-list-comprehension) (<https://neo4j.com/docs/developer-manual/current/cypher/syntax/lists/#cypher-list-comprehension>) generated from schema is now used.

It means that, instead of loading blindly object graphs up to a certain depth, OGM will now only query the nodes and relationships that are expressed in your domain model. For applications that do not require following all the relationships, this can be a huge performance improvement.

However, this new load strategy does not support unlimited depth (depth = -1). Either use specific depth or use `PATH_BASED_LOAD_STRATEGY`, to revert to the old strategy.

```
session.setLoadStrategy(LoadStrategy.PATH_LOAD_STRATEGY);
```



Default depth for `Session.load*` methods without depth parameter is 1, so there is no need to change anything. Only calls with explicit `depth = -1` need to be changed.

Migration checklist

Here are the things you (may) have to adapt :

- Id handling : `Long` native ids are not mandatory anymore. See [GraphId field](#).
- Primary indexes are now deprecated and replaced by `@Id` See [Entity identifier](#).
- Annotations on accessors are no longer valid. See [Annotating entities](#).
- Loading with depth `-1` calls have to be reviewed (see above).
- The `ogm.properties` file and environment variable have been removed. You now have to provide explicitly the configuration file or configure programmatically. See [the configuration section](#).
- The driver class name in the configuration is now inferred from connection URL.
- Java 8 is now mandatory. OGM won't run on any previous Java version.
- Neo4j 3.1 or higher is required, because of the new schema load strategy, which uses list comprehensions.
- Java 8 dates are now better supported ; the use of `java.util.Date` or converters is not required anymore. You may want to switch to more fine grained date types like `Instant`. See [conversions](#).
- The query filters are now immutable. See [Filters](#).

Appendix B: 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.

Appendix C: Frequently Asked Questions (FAQ)

What is the difference between Neo4j OGM and Spring Data Neo4j (SDN)?

Spring Data Neo4j (SDN) uses the OGM under the covers. It's like Spring Data JPA, where JPA/Hibernate underly it. Most of the power of SDN actually comes from the OGM.

How are labels generated when using inheritance?

All concrete classes generate a label, abstract classes and interfaces not. If any kind of class or interface gets annotated with `@NodeEntity` or `@NodeEntity(label="customLabel")` it will generate a label. Any class annotated with `@Transient` will not generate a label.

License

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

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.

NonCommercial

You may not use the material for commercial purposes.

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 <https://creativecommons.org/licenses/by-nc-sa/4.0/> for further details. The full license text is available at <https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>.