Good Relationships

The Spring Data Neo4j Guide Book

Michael Hunger, Oliver Gierke, Vince Bickers, Adam George, Luanne Misquitta, Michal Bachman

Version 4.1.2.RELEASE, 2016-06-15

Table of Contents

Spring Data Neo4j Version 4.1 (April 2016)

Preface

Chapter 1. About this guide book

1.1. The Spring Data Neo4j Project

Welcome to the Spring Data Neo4j Guide Book. Thank you for taking the time to get an in-depth look into Spring Data Neo4j. This project is part of the Spring Data project, which brings the convenient programming model of the Spring Framework to modern NOSQL databases. Spring Data Neo4j, as the name alludes to, aims to provide support for the graph database Neo4j.

1.2. Feedback

It was written by developers for developers. Hopefully we've created a guide that is well-received by our peers.

If you have any feedback on Spring Data Neo4j or this book, please provide it via the SpringSource JIRA, StackOverflow or the Neo4j Google Group.

1.3. Format of the Book

This book is a classic reference document, containing detailed information about the library. It discusses the programming model, the underlying assumptions, and internals, as well as the APIs for the object-graph mapping. The reference documentation is typically used to look up particular pieces of information or to drill down into certain topics.

1.4. Acknowledgements

We would like to thank everyone who contributed to this book, especially Oliver Gierke, the lead of the Spring Data Project.

Many thanks to our colleagues at Neo Technology and Graph Aware who not only contributed to Spring Data Neo4j but also provided content and feedback for this book.

We also appreciate very much the foresight of Rod Johnson and Emil Eifrem to initiate the original project. Their leadership inspired collaboration between the engineering teams at SpringSource and Neo Technology, a tremendous help during the making of Spring Data Neo4j.

Last but not least we thank our vibrant community for giving us feedback, reporting issues and suggesting improvements. Without that important feedback we wouldn't be where we are today.

Enjoy the book!

Chapter 2. Introduction to Neo4j

2.1. 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 support keeping 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.

2.2. About Neo4j

Neo4j 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.
- 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

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.

2.3. Querying the Graph with Cypher

Neo4j provides a graph query language called Cypher which draws from many sources. It resembles SQL clauses but is centred around matching iconic representation of patterns in the graph.

Cypher queries typically begin with a `MATCH` clause, which can be used to provide a way to pattern match against the graph. Match clauses can introduce new identifiers for nodes and relationships. In the `WHERE` clause additional filtering of the result set is applied by evaluating expressions. The `RETURN` clause defines which part of the query result will be available to the caller. Aggregation also happens in the return clause by using aggregation functions on some of the returned values. Sorting can happen in the `ORDER BY` clause and the `SKIP` and `LIMIT` parts restrict the result set to a certain window.

Cypher statements are executed against Neo4j Server using an HTTP-based protocol, which is utilised by Spring Data Neo4j.

Cypher Examples on the Movies Dataset

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

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

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

Cypher Examples on the Movies Dataset

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

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

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

2.4. Indexing

Neo4j's schema indexes are used automatically by Cypher when set up in your database. Spring Data Neo4j (version 4) does not provide facilities for handling that setup out of the box. This is a seeding, migration or maintenance effort handled by the group responsible for the database maintenance.

Spring Data Neo4j 4.1 Reference Documentation

This part of the Spring Data Neo4j (SDN) Guide book provides the reference documentation for SDN 4.1.

Its content covers information about the programming model, APIs, concepts, annotations and technical details of Spring Data Neo4j, version 4.1.

Whenever you look for the means to employ the full power of the Spring Data Neo4j library, you should be able to find your answers in this reference section. If you don't, please inform us about missing or incorrect content.

Chapter 3. About the Spring Data Project

Spring Data is a SpringSource project that aims to provide Spring's convenient programming model and well known conventions for NOSQL databases. Currently there is support for graph (Neo4j), key-value (Redis), document (MongoDB) and relational (JPA) databases.

The Spring Data Neo4j project, as part of the Spring Data initiative, aims to simplify development with the Neo4j graph database. Like JPA, it uses annotations on simple POJO domain objects. Together with metadata, the annotations drive mapping the POJO entities and their fields to nodes, relationships, and properties in the graph database.

3.1. About Spring Data Neo4j 4

For version 4.0, Spring Data Neo4j was rewritten from scratch to natively support Neo4j deployments in standalone server mode. It uses Cypher, the Neo4j query language, and the HTTP protocol to communicate with the database. It's therefore worth noting that there **will be backward compatibility issues** when migrating to version 4.x, so be sure to check the Migration Guide to avoid any unwanted surprises.

Version 4.1 introduces support for connecting to an embedded instance of Neo4j.

For integration of Neo4j and other languages, please see Language Guides.

Chapter 4. Overview

The explanation of Spring Data Neo4j's programming model starts with some underlying details. The basic concepts of the Object-Graph Mapping (OGM) library used by Spring Data Neo4j internally, is explained in the initial chapter.

4.1. Getting Started

To get started with a simple application, you need only your domain model and (optionally) the annotations (see Defining node entities) 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.

4.2. Adding Neo4j Graph Queries

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

4.3. Managing Relationships

Relationships between entities are first class citizens in a graph database and therefore worth a separate chapter (Relating Node Entities) describing their usage in Spring Data Neo4j.

4.4. Repositories

Spring Data Commons provides a very powerful repository infrastructure that is also leveraged in Spring Data Neo4j. Those repositories consist of a composition of interfaces that declare the available functionality in each repository. The implementation details of commonly-used persistence methods are handled by the library, which makes them very convenient for typical CRUD and query operations. The repositories are extensible by annotated, named or derived finder methods (like in (G)Rails). For custom implementations of repository methods you are free to add your own code. (CRUD with repositories).

4.5. Neo4jTemplate

Being a Spring Data library, Spring Data Neo4j offers a Neo4jTemplate for interacting with the mapped entities and the Neo4j graph database if you don't want to use repositories. As of version 4, `Neo4jTemplate` is based on the `org.neo4j.ogm.session.Session` object and adds mostly exception translation around the functionality. Support for Spring Data Neo4j Repositories is also based on `Neo4jTemplate`, so the underlying functionality is identical.

In keeping with the Spring Data ethos, an operations interface is used to define the API exposed by the `Neo4jTemplate`. This interface is called `Neo4jOperations`, the default implementation of which is `Neo4jTemplate`. Users of the framework from version 4 are encouraged to code against `Neo4jOperations` instead of the `Neo4jTemplate` directly.

4.6. Mapping Strategies

Because Neo4j is a schema-free database, Spring Data Neo4j uses a simple mechanism to map Java types to Neo4j nodes using labels. How that works is explained here: Simplified Object Graph Mapping.

4.7. Transactional Support

Neo4j uses transactions to guarantee the integrity of your data and Spring Data Neo4j supports this fully. The implications of this are described in the chapter around Transactions.

4.8. Configuration

Currently, only Java bean-based configuration is supported. See Getting started for more details.

4.9. Performance

Spring Data Neo4j 4 has been rebuilt from the ground up with performance in mind. More information can be found in Performance Considerations.

Chapter 5. Getting started

Spring Data Neo4j 4.1 (SDN 4.1) dramatically simplifies development, but some setup is naturally required. For building the application, your build automation tool needs to be configured to include the Spring Data Neo4j dependencies and after the build setup is complete, the Spring application needs to be configured to make use of Spring Data Neo4j. Examples for these different setups can be found in the Spring Data Neo4j examples.

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

5.1. Dependencies for Spring Data Neo4j 4.1

Maven dependencies

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j</artifactId>
    <version>{version}</version>
</dependency>
```

By default, SDN 4.1 will use the Http driver to connect to Neo4j and you don't need to declare it as a separate dependency in your pom. If you want to use the embedded driver in your production application, you must add the following dependency as well. (This dependency is not required if you only want to use the embedded driver for testing. See the section on Testing below for more information).

```
<!-- add this dependency if you want to use the embedded driver -->
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-ogm-embedded-driver</artifactId>
    <version>{ogm-version}</version>
</dependency>
```

Gradle dependencies

```
dependencies {
    compile 'org.springframework.data:spring-data-neo4j:{version}'
    # add this dependency if you want to use the embedded driver
    compile 'org.neo4j:neo4j-ogm-embedded-driver:{ogm-version}'
}
```

Ivy dependencies

```
<dependency org="org.springframework.data" name="spring-data-neo4j" rev="{version}"/>
<!-- add this dependency if you want to use the embedded driver -->
<dependency org="org.neo4j" name="neo4j-ogm-embedded-driver" rev="{ogm-version}"/>
```

5.2. Spring configuration

Users of SDN 4.1 can configure their applications using Java-based bean configuration.

5.2.1. Java-based bean configuration

We recommend that your Spring context should extend the core `Neo4jConfiguration` class that comes with Spring Data Neo4j. The example below shows how this can be done.

NOTE	You will need to override `getSessionFactory()` and `getSession()` bean definitions to provide the required context for your own application. This is explained in more detail below.

Pure Java Configuration

```
@Configuration
@EnableNeo4jRepositories(basePackages = "org.neo4j.example.repository")
@EnableTransactionManagement
public class MyConfiguration extends Neo4jConfiguration {

    @Bean
    public SessionFactory getSessionFactory() {
        // with domain entity base package(s)
        return new SessionFactory("org.neo4j.example.domain");
    }

    // needed for session in view in web-applications
    @Bean
    @Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
    public Session getSession() throws Exception {
        return super.getSession();
    }

}
```

5.3. Drivers

SDN 4.1 now provides support for connecting to Neo4j using different drivers. As a result, the `RemoteServer` and `InProcessServer` classes from previous versions should not be used, and are no longer supported.

The following drivers are available.

- Http driver
- Embedded driver

By default, SDN 4.1 will try to configure the driver from a file `ogm.properties`, which it expects to find on the classpath. In many cases you won't want to, or will not be able to provide configuration

information via a properties file. In these cases you can configure your application programmatically instead, using a `Configuration` bean.

The following sections describe how to setup Spring Data Neo4j 4.1 using both techniques.

Configuring the Http Driver

The Http Driver connects to and communicates with a Neo4j server over Http. An Http Driver must be used if your application is running in client-server mode.

NOTE	The Http Driver is the default driver for SDN 4.1, and doesn't need to be explicitly declared in your pom file.

Properties file

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

NOTE	SDN expects the properties file to be called "ogm.properties". If you want to configure your application using a *different* properties file, you must either set a System property or Environment variable called "ogm.properties" pointing to the alternative configuration file you want to use.

Java Configuration

To configure the Driver programmatically, create a Configuration bean and pass it as the first argument to the SessionFactory constructor in your Spring configuration:

```
import org.neo4j.ogm.config.Configuration;
...

@Bean
public Configuration getConfiguration() {
   Configuration config = new Configuration();
   config
       .driverConfiguration()
       .setDriverClassName("org.neo4j.ogm.drivers.http.driver.HttpDriver")
       .setURI("http://user:password@localhost:7474");
   return config;
}

@Bean
public SessionFactory getSessionFactory() {
    return new SessionFactory(getConfiguration(), <packages> );
}
```

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

Configuring the Embedded Driver

The Embedded Driver connects directly to the Neo4j database engine. There is no server involved, therefore no network overhead between your application code and the database. You should use the Embedded driver if you don't want to use a client-server model, or if your application is running as a Neo4j Unmanaged Extension.

If you want to use the Embedded driver in your production application, you will need to explicitly declare the required driver dependency in your project's pom file:

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

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.

Properties file (permanent data store)

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

Properties file (impermanent data store)

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

Java Configuration (permanent data store)

The same technique is used for configuring the Embedded driver as for the Http Driver. Set up a Configuration bean and pass it as the first argument to the SessionFactory constructor:

```
import org.neo4j.ogm.config.Configuration;
...

@Bean
public Configuration getConfiguration() {
   Configuration config = new Configuration();
   config
       .driverConfiguration()
       .setDriverClassName("org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver")
       .setURI("file:///var/tmp/graph.db");
   return config;
}

@Bean
public SessionFactory getSessionFactory() {
    return new SessionFactory(getConfiguration(), <packages> );
}
```

If you want to use an impermanent data store simply omit the URI attribute from the Configuration:

```
@Bean
public Configuration getConfiguration() {
   Configuration config = new Configuration();
   config
       .driverConfiguration()
       .setDriverClassName("org.neo4j.ogm.drivers.embedded.driver.EmbeddedDriver");
   return config;
}
```

Authentication

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

Properties file options:

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

# as separate attributes
username="user"
password="password"
```

Java Configuration options

```
// embedded in the driver URI
@Bean
public Configuration getConfiguration() {
   Configuration config = new Configuration();
   config
       .driverConfiguration()
       .setDriverClassName("org.neo4j.ogm.drivers.http.driver.HttpDriver")
       .setURI("http://user:password@localhost:7474");
   return config;
}

// separately, as plain text credentials
@Bean
public Configuration getConfiguration() {
   Configuration config = new Configuration();
   config
       .driverConfiguration()
       .setDriverClassName("org.neo4j.ogm.drivers.http.driver.HttpDriver")
       .setCredentials("user", "password")
       .setURI("http://localhost:7474");
   return config;
}


// using a Credentials instance:

@Bean
public Credentials credentials() {
    return new UsernameAndPasswordCredentials(...);
}

@Bean
public Configuration getConfiguration() {
   Configuration config = new Configuration();
   config
       .driverConfiguration()
       .setDriverClassName("org.neo4j.ogm.drivers.http.driver.HttpDriver")
       .setCredentials(credentials())
       .setURI("http://localhost:7474");
   return config;
}
```

NOTE Currently only Basic Authentication is supported by Neo4j, so the only Credentials implementation available is `UsernameAndPasswordCredentials`

5.3.4. Testing

Maven dependencies for testing SDN 4.1 applications

```
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-neo4j</artifactId>
            <version>${sdn.version}</version>
            <type>test-jar</type>
        </dependency>

        <!-- the neo4j-ogm-test jar provides access to the http and embedded drivers
for testing purposes -->
        <dependency>
           <groupId>org.neo4j</groupId>
           <artifactId>neo4j-ogm-test</artifactId>
           <version>${neo4j-ogm.version}</version>
           <type>test-jar</type>
           <scope>test</scope>
       </dependency>

        <dependency>
            <groupId>org.neo4j</groupId>
             <artifactId>neo4j-kernel</artifactId>
             <version>${neo4j.version}</version>
             <type>test-jar</type>
        </dependency>

        <dependency>
             <groupId>org.neo4j.app</groupId>
             <artifactId>neo4j-server</artifactId>
             <version>${neo4j.version}</version>
             <type>test-jar</type>
        </dependency>

       <dependency>
           <groupId>org.neo4j.test</groupId>
           <artifactId>neo4j-harness</artifactId>
           <version>${neo4j.version}</version>
           <scope>test</scope>
       </dependency>
```

NOTE

In SDN 4.1, the `InProcessServer` has been deprecated. This class was used in previous versions to set up an in-memory Http server so that you could run your tests. This is no longer appropriate given the new Driver mechanism, and we recommend you configure an Embedded Driver (impermanent data store) for your integration tests instead.

5.3.5. SessionFactory Bean

The `SessionFactory` is needed by SDN 4.1 to create instances of `org.neo4j.ogm.session.Session` as required. When constructed, it sets up the object-graph mapping metadata, which is then used across all `Session` objects that it creates. As seen in the above example, the packages to scan for domain object metadata should be provided to the `SessionFactory` constructor.

Note that the session factory should typically be application-scoped. While you can use a narrower scope for this if you like, there is typically no advantage in doing so.

5.3.6. Session Bean

A `Session` is used to drive the object-graph mapping framework on which Spring Data Neo4j is based. All repository implementations and `Neo4jTemplate` are driven by the `Session`. You can also auto-wire it into your Spring beans and code against it directly if you wish.

The life cycle of a `Session` is important to consider because it keeps track of the changes that have been made to entities and their relationships. The reason it does this is so that only entities and relationships that have changed get persisted on save, which is particularly efficient when working with large graphs. Note, however, that the `Session` **does not ever return cached objects** so there's no risk of getting stale data on load; it always hits the database.

If your application relies on long-running sessions and does not reload entities then you may not see changes made from other users and find yourself working with outdated objects. On the other hand, if your sessions have too narrow a scope then your save operations can be unnecessarily expensive, as updates will be made to all objects if the session isn't aware of the those that were originally loaded.

There's therefore a trade off between the two approaches. In general, the scope of a `Session` should correspond to a "unit of work" in your application. What this means depends on the usage scenario, but in a typical web-based Spring application we recommend using a request-scoped or HTTP-session-scoped `Session`. Either way, if you make sure you load fresh data at the beginning of each unit of work then data integrity shouldn't be a problem.

Additional beans can be configured just by defining them in the Spring context in the normal way.

Chapter 6. Programming model

This chapter covers the fundamentals of the programming model behind Spring Data Neo4j, version 4. It discusses the mapping mode, the annotations provided by Spring Data Neo4j and how to use them.

6.1. Under the hood

6.1.1. Metadata collection

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

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

6.1.2. The Session object

The Spring repositories and `Neo4jTemplate` are both backed by `org.neo4j.ogm.session.Session`, which is a key component of the framework. The Session provides methods to load, save or delete object graphs from the database and also provides transaction support. The new `Neo4jTemplate` is essentially a wrapper around this `Session`, which exposes all of its useful methods but handles transactions and provides the traditional Spring Data operations and exception translation.

6.1.3. Explicit save

Unlike the previous AspectJ-driven mapping, Spring Data Neo4j 4 doesn't automatically commit when a transaction closes, so an explicit call to `save(⋯)` is required in order to persist changes to the database.

6.1.4. Fine-grained control via depth specification

The new object mapping framework in Spring Data Neo4j introduces 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 repository and template methods, but SDN 4 chooses sensible defaults so that you don't have to specify the depth attribute unless you want change the default values.

Default depth for loading

By default, loading an instance will map that object's simple properties and its immediately-related objects (i.e. depth = 1). This helps to avoid accidentally loading the entire graph into memory, but allows a single request to fetch not only the object of immediate interest, but also its closest

neighbours, which are likely also to be of interest. This strategy attempts to strike a balance between loading too much of the graph into memory and having to make repeated requests for data.

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

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

Default depth for persisting

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

6.2. Simplified Object-Graph Mapping

As of version 4, Spring Data Neo4j supports mapping annotated and non-annotated object models. It's possible to save any POJO without annotations to the graph, as the framework applies conventions to decide what to do. This is useful in cases when you don't have control over the classes that you want to persist. The recommended approach, however, is to use annotations wherever possible, since this gives greater control and means that code can be refactored safely without risking breaking changes to the labels and relationships in your graph.

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

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

The object graph mapping comes into play whenever an entity is constructed from a node or relationship. This could be done explicitly during the lookup or create operations of the repositories and the `Neo4jTemplate` but also implicitly while executing any graph operation that returns nodes or relationships and expecting mapped entities to be returned.

Entities handled by Spring Data Neo4j must have an empty constructor to allow the library to construct the objects.

Unless annotations are used to specify otherwise, the framework will attempt to map any of an object's "simple" fields to node properties and any rich composite objects to related nodes. A "simple" field is any primitive, boxed primitive or String or arrays thereof, essentially anything that naturally fits into a Neo4j node property. For related entities the type of a relationship is inferred by the bean property name, as outlined in the examples below.

6.3. Defining node entities

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

6.3.1. @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 Spring Data Neo4j must have a zero-argument constructor to allow the library to construct the objects, although it doesn't need to be declared public.

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. This replaces the previous `@TypeAlias` annotation. The default label is just the simple class name of the annotated entity. All parent classes are also added as labels so that retrieving a collection of nodes via a parent type is supported.

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

Persisting an annotated entity

```
@NodeEntity
public class Actor extends DomainObject {

   @GraphId
   private Long id;

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

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

}

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

   @GraphId Long id;

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

}
```

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

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

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

Persisting a non-annotated entity

```
public class Actor extends DomainObject {

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

}

public class Movie {

   private Long id;
   private String name;

}
```

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

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

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

6.3.2. @GraphId: Neo4j ID Field

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

NOTE	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 a node with id 0.

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

Entity Equality

Entity equality can be a grey area, and it is debatable whether natural keys or database IDs best describe equality, as there is the issue of versioning over time, etc. In previous versions of Spring Data Neo4j, it was recommended to honour the convention that database-issued IDs are the basis for equality, despite the consequences.

In version 4, the dependency of the framework upon a particular style of `equals()` or `hashCode()` implementation has been removed. The graph-id field is directly checked to see if two entities

represent the same node and a 64-bit hash code is used for dirty checking, so you're not forced to write your code in a certain way!

However, we do think it's important to mention that if you use the `@GraphId` field in your `hashCode()` method then this comes with a caveat. When you first persist an entity, its hashcode changes because Spring Data Neo4j populates the database ID on save.

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

6.3.3. @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 declared with the `transient` modifier are exempted from persistence. All fields that contain primitive values are persisted directly to the graph. All fields convertible to a `String` using the Spring conversion services will be stored as a string. Spring Data Neo4j includes default type converters that deal with the following types:

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

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

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

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

6.4. Relating Node Entities

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

6.4.1. @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 Spring Data Neo4j 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). These fields come in two forms, modifiable or read-only. Modifiable fields are of the type `Collection<T>`, and read-only fields are `Iterable<T>`, where T is a type annotated with `@NodeEntity`.

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()`. By default, the *related* node or relationship entities will just be loaded to minimum depth 0, which means their properties will be set but no further 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..

Spring Data Neo4j 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_OF` 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.

NOTE	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.

6.4.2. @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 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. Currently, the `type` must always be specified on the `@RelationshipEntity` annotation.

NOTE	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 {
    @GraphId   private Long relationshipId;
    @Property  private String title;
    @StartNode private Actor actor;
    @EndNode   private Movie movie;
}

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

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

Additionally, SDN4 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 SDN 4.

In previous versions of Spring Data Neo4j, a dynamic relationship type field was supported. However, this has been dropped completely for version 4, since it was not possible to manage it effectively for both reading from and writing to the graph.

NOTE	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.

6.4.3. Discriminating Relationships Based on End Node 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;
...
}
```

In previous versions of Spring Data Neo4j, you would have to add an `enforceTargetType` attribute into every clashing `@Relationship` annotation for this to map correctly. Thanks to changes in the underlying object-graph mapping mechanism, this is no longer necessary and the above will work just fine.

However, please be aware that this will only work 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.

6.4.4. Ambiguity in relationships

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

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

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

6.5. Indexing

Indexing is used in Neo4j to quickly find nodes and relationships from which to start graph operations. Indexes are also employed to ensure uniqueness of elements with certain labels and properties.

NOTE	Please note that the lucene-based manual indexes are deprecated with Neo4j 2.0. The default index is now based on labels and schema indexes and the related old APIs have been deprecated as well. The "legacy" index framework should only be used for fulltext and spatial indexes which are not currently supported via schema-based indexes.

6.5.1. Index Management in Spring Data Neo4j 4

In Spring Data Neo4j 4, index management concerns were removed from the mapping framework entirely. Index creation and management is therefore now outside the scope of this document. Please see the Neo4j documentation on indexes for for information: http://neo4j.com/docs/stable/query-schema-index.html

6.5.2. Index queries in Neo4jTemplate

Schema indexes are automatically used by Neo4j's Cypher engine, so using the annotated or derived repository finders or the query methods in `Neo4jTemplate` will use them out of the box.

6.5.3. Neo4j Auto Indexes

Neo4j allows to configure (legacy) auto-indexing for certain properties on nodes and relationships. It is possible to use the specific index names `node_auto_index` and `relationship_auto_index` when querying indexes in Spring Data Neo4j either with the query methods in template and repositories or via Cypher.

6.5.4. Full-Text Indexes

Previous versions of Spring Data Neo4j offered support for full-text queries using the manual index facilities. However, as of SDN 4, this is no longer supported.

To create fulltext entries for an entity you can add the updated nodes within `AfterSaveEvents` to a remote fulltext-index via Neo4j's REST API. If you use Http Driver and the `HttpRequest` used by the OGM, then authentication will be taken care of as well.

Indexing Persons upon persistence with the HTTP Driver

```
final CloseableHttpClient httpClient = HttpClients.createDefault();

@Bean
ApplicationListener<AfterSaveEvent> afterSaveEventApplicationListener() {
    return new ApplicationListener<AfterSaveEvent>() {
        @Override
        public void onApplicationEvent(AfterSaveEvent event) {
            if(event.getEntity() instanceof Person) {
                String uri = Components.driver().getConfiguration().getURI() +
                            "/db/data/index/node/" + indexName;
                HttpPost httpPost = new HttpPost(uri);
                Person person = (Person) event.getEntity();
                //Construct the JSON statements
                try {
                    httpPost.setEntity(new StringEntity(json.toString()));
                    HttpRequest.execute(httpClient, httpPost,
                                        Components.driver().getConfiguration()
.getCredentials());
                } catch (Exception e) {
                    //Handle any exceptions
                }
            }
        }
    };
}
```

Indexing Persons upon persistence with the Embedded Driver

```
@Bean
ApplicationListener<AfterSaveEvent> afterSaveEventApplicationListener() {
    return new ApplicationListener<AfterSaveEvent>() {
        @Override
        public void onApplicationEvent(AfterSaveEvent event) {

            if(event.getEntity() instanceof Person) {
                EmbeddedDriver embeddedDriver = (EmbeddedDriver) Components.driver();
                GraphDatabaseService databaseService = embeddedDriver
.getGraphDatabaseService();
                Person person = (Person) event.getEntity();
                try (Transaction tx = databaseService.beginTx()) {
                    Node node = databaseService.getNodeById(person.getNodeId());
                    databaseService.index().forNodes(indexName).add(node, key, value);
                    tx.success();
                }
            }
        }
    };
}
```

Fulltext query support is still available via Cypher queries which can be executed via the `Session` or `Neo4jTemplate`, or as a `@Query` defined in a repository class.

6.5.5. Spatial Indexes

Previous versions of Spring Data Neo4j offered support for spatial queries using the `neo4j-spatial` library. However, as of SDN 4 at least, this is no longer supported.

A strategy similar to the full-text indexes being updated within `AfterSaveEvents` can be employed to support Spatial Indexes. The Neo4j Spatial Plugin exposes a REST API to interact with the library.

6.6. Neo4jTemplate

The `Neo4jTemplate` offers the convenient API of Spring templates for the Neo4j graph database. As of version 4, the Spring Data Neo4j Template wraps the underlying object-graph mapping `Session`, but still provides the core functionality to persist objects to the graph and load them in a variety of ways. Indeed, you can just use the `Session` directly in your code if you need greater control, but the `Neo4jTemplate` may well be easier for general use.

6.6.1. Basic Operations

For Spring Data Neo4j 4, the changes to the underlying architecture have led to the pruning of the `Neo4jTemplate` feature set. Basic operations are now entirely limited to CRUD operations on entities and executing arbitrary Cypher queries; more low-level manipulation of the graph database is not possible.

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

Given that the latest version of the framework is driven by Cypher queries alone, there's no way to work directly with `Node` and `Relationship` objects any more in remote server mode. Similarly, the `traverse()` method has disappeared, 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.

6.6.2. Entity Persistence

`Neo4jTemplate` allows you to `save`, `load`, `loadAll` and `delete` entities. However, as of SDN 4, it no longer provides the stored type information via `getStoredJavaType`. The eagerness with which objects are retrieved is controlled by specifying the 'depth' argument to any of the load methods.

All of these basic CRUD methods just call onto the underlying methods of `Session`, albeit with transaction handling and exception translation managed for you.

6.6.3. Cypher Queries

The `Neo4jTemplate` also allows execution of arbitrary Cypher queries via its `query`, `queryForObject` and `queryForObjects` methods. Cypher queries that return tabular results should be passed into the `query` method. An `org.neo4j.ogm.session.result.Result` is returned. This consists of `org.neo4j.ogm.session.result.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.

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

6.6.4. Transactions

The `Neo4jTemplate` provides implicit transactions for some of its methods. For instance `save`, `delete` and `query` provide auto-commit transactions. For other modifying operations you would need to provide Spring Transaction management using `@Transactional` or the `TransactionTemplate`.

6.6.5. Data Manipulation Events (formerly Lifecycle Events)

Neo4j Template utilises Spring's event mechanism through `ApplicationListener` and `ApplicationEvent` to notify interested parties about certain data manipulations performed through it. The following hooks are available in the form of types of application event:

- BeforeSaveEvent
- AfterSaveEvent
- BeforeDeleteEvent
- AfterDeleteEvent

The point at which these events get fired by the `Neo4jTemplate` should be pretty obvious from the names. The following example demonstrates how to hook into the Spring application events API and register listeners that perform behaviour across types of entities as actions are performed using the template.

Auditing Entities and Generating Unique Application-level IDs

```
@Configuration
@EnableNeo4jRepositories
public class ApplicationConfig extends Neo4jConfiguration {
    ...
    @Bean
    ApplicationListener<BeforeSaveEvent> beforeSaveEventApplicationListener() {
        return new ApplicationListener<BeforeSaveEvent>() {
            @Override
            public void onApplicationEvent(BeforeSaveEvent event) {
                AcmeEntity entity = (AcmeEntity) event.getEntity();
                entity.setUniqueId(acmeIdFactory.create());
            }
        };
    }

    @Bean
    ApplicationListener<AfterSaveEvent> afterSaveEventApplicationListener() {
        return new ApplicationListener<AfterSaveEvent>() {
            @Override
            public void onApplicationEvent(AfterSaveEvent event) {
                AcmeEntity entity = (AcmeEntity) event.getEntity();
                auditLog.onEventSaved(entity);
            }

    }

    @Bean
    ApplicationListener<AfterDeleteEvent> deleteEventApplicationListener() {
        return new ApplicationListener<AfterDeleteEvent>() {
            @Override
            public void onApplicationEvent(AfterDeleteEvent event) {
                AcmeEntity entity = (AcmeEntity) event.getEntity();
                auditLog.onEventDeleted(entity);
            }
        };
    }
    ...
```

Note that changes made to entities in the before-save event application listener are reflected in the stored entity - after-save ones are not.

6.7. CRUD with repositories

The repositories provided by Spring Data Neo4j build on the composable repository infrastructure in Spring Data Commons. These allow for interface-based composition of repositories consisting of provided default implementations for certain interfaces and additional custom implementations for other methods.

Spring Data Neo4j comes with a single `org.springframework.data.repository.PagingAndSortingRepository` specialisation called `GraphRepository<T>` used for all object-graph mapping repositories. This sub-interface also adds specific finder methods that take a *depth* argument to control the horizon with which related entities are fetched and saved. Generally, it provides all the desired repository methods. If other operations are required then the additional repository interfaces should be added to the individual interface declaration.

NOTE	`GraphRepository` no longer combines `IndexRepository` and `TraversalRepository` because, for reasons explained earlier, these features are no longer supported in Spring Data Neo4j as of version 4.

6.7.1. GraphRepository

As of SDN 4, this `GraphRepository<T>` should be the interface from which your entity repository interfaces inherit, with `T` being specified as the domain entity type to persist.

Examples of methods you get for free out of `GraphRepository` are as follows. For all of these examples the ID parameter is a `Long` that matches the graph ID:

Load an entity instance via an id
`T findOne(id)`

Check for existence of an id in the graph
`boolean exists(id)`

Iterate over all nodes of a node entity type
`Iterable<T> findAll()` `Iterable<T> findAll(Sort …)` `Page<T> findAll(Pageable …)`

Count the instances of the repository entity type
`Long count()`

Save entities
`T save(T)` and `Iterable<T> save(Iterable<T>)`

Delete graph entities
`void delete(T)`, `void delete(Iterable<T>)`, and `void deleteAll()`

6.7.2. Query and Finder Methods

Annotated queries

Queries using the Cypher graph query language can be supplied with the `@Query` annotation.

That means a repository method annotated with
`@Query("MATCH (:Actor {name:{name}})-[:ACTED_IN]→(m:Movie) return m")`
will use the supplied query query to retrieve data from Neo4j.

The named or indexed parameter `{param}` will be substituted by the actual method parameter. Node

and Relationship-Entities are handled directly and converted into their respective ids, Iterables thereof as well. All other parameters types are provided directly (i.e. Strings, Longs, etc).

NOTE	In the current version, custom queries do not support paging, sorting or a custom depth. `@Query` 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.

Query results

Typical results for queries are `Iterable<Type>`, `Iterable<Map<String,Object>>` or simply `Type`. Nodes and relationships are converted to their respective entities (if they exist). Other values are converted using the registered conversion services (e.g. enums).

Cypher examples

`MATCH (n) WHERE id(n)=9 RETURN n`

returns the node with id 9

`MATCH (movie:Movie {title:'Matrix'}) RETURN movie`

returns the nodes which are indexed with title equal to 'Matrix'

`MATCH (movie:Movie {title:'Matrix'})<-[:ACTS_IN]-(actor) RETURN actor.name`

returns the names of the actors that have a ACTS_IN relationship to the movie node for 'Matrix'

```
MATCH (movie:Movie {title:'Matrix'})<-[r:RATED]-(user) WHERE r.stars > 3 RETURN user.name,
r.stars, r.comment
```

returns users names and their ratings (>3) of the movie titled 'Matrix'

```
MATCH (user:User {name='Michael'})-[:FRIEND]-(friend)-[r:RATED]->(movie) RETURN movie.title,
AVG(r.stars), COUNT(*) ORDER BY AVG(r.stars) DESC, COUNT(*) DESC
```

returns the movies rated by the friends of the user 'Michael', aggregated by `movie.title`, with averaged ratings and rating-counts sorted by both

Examples of Cypher queries placed on repository methods with @Query where values are replaced with method parameters, as described in the Annotated queries) section.

```
public interface MovieRepository extends GraphRepository<Movie> {

    // returns the node with id equal to idOfMovie parameter
    @Query("MATCH (n) WHERE id(n)={0} RETURN n")
    Movie getMovieFromId(Integer idOfMovie);

    // returns the nodes which have a title according to the movieTitle parameter
    @Query("MATCH (movie:Movie {title={0}}) RETURN movie")
    Movie getMovieFromTitle(String movieTitle);

    // returns the Actors that have a ACTS_IN relationship to the movie node with the
title equal to movieTitle parameter.
    // (The parenthesis around 'movie' and 'actor' in the match clause are optional.)
    @Query("MATCH (movie:Movie {title={0}})<-[:ACTS_IN]-(actor) RETURN actor")
    Page<Actor> getActorsThatActInMovieFromTitle(String movieTitle, PageRequest page);

    // returns users who rated a movie (movie parameter) higher than rating (rating
parameter)
    @Query("MATCH (movie:Movie)<-[r:RATED]-(user) " +
           "WHERE id(movie)={movieId} AND r.stars > {rating} " +
           "RETURN user")
    Iterable<User> getUsersWhoRatedMovieFromTitle(@Param("movieId") Movie movie,
@Param("rating") Integer rating);

    // returns users who rated a movie based on movie title (movieTitle parameter)
higher than rating (rating parameter)
    @Query("MATCH (movie:Movie {title:{0}})<-[r:RATED]-(user) " +
           "WHERE r.stars > {1} " +
           "RETURN user")
     Iterable<User> getUsersWhoRatedMovieFromTitle(String movieTitle, Integer rating);
}
```

Queries derived from finder-method names

Using the metadata infrastructure in the underlying object-graph mapper, a finder method name can be split into its semantic parts and converted into a cypher query. Navigation along relationships will be reflected in the generated `MATCH` clause and properties with operators will end up as expressions in the `WHERE` clause. The parameters will be used in the order they appear in the method signature so they should align with the expressions stated in the method name.

Some examples of methods and corresponding Cypher queries of a PersonRepository

```
public interface PersonRepository extends GraphRepository<Person> {

    // MATCH (person:Person {name={0}}) RETURN person
    Person findByName(String name);

    // MATCH (person:Person)
    // WHERE person.age = {0} AND person.married = {1}
    // RETURN person
    Iterable<Person> findByAgeAndMarried(int age, boolean married)

}
```

NOTE In the current version, derived finders do not support paging, sorting or a custom depth.

6.7.3. Creating repositories

The `Repository` instances are only created through Spring and can be auto-wired into your Spring beans as required.

Using basic GraphRepository CRUD-methods

```
@Repository
public interface PersonRepository extends GraphRepository<Person> {}

public class MySpringBean {
   @Autowired
   private PersonRepository repo;
   ...
}

// then you can use the repository as you would any other object
Person michael = repo.save(new Person("Michael", 36));

Person dave = repo.load(123);

long numberOfPeople = repo.count();
```

The recommended way of providing repositories is to define a repository interface per domain class. The underlying Spring repository infrastructure will automatically detect these repositories, along with additional implementation classes, and create an injectable repository implementation to be used in services or other spring beans.

Example Spring configuration bean

```
@Configuration
@ComponentScan({"com.example.sdn"})
@EnableNeo4jRepositories("com.example.sdn.repo")
@EnableTransactionManagement
public class PersistenceContext {

   @Bean
   public SessionFactory getSessionFactory() {
      return new SessionFactory("com.example.sdn.domain");
   }
   // more bean definition methods here
}
```

6.8. Conversion

The object-graph mapping framework on which Spring Data Neo4j is built 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.

6.8.1. Built-In Type Conversions

By default, Spring Data Neo4j will automatically perform the following type conversions:

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

Two Date converters are provided "out of the box"

1. @DateString
2. @DateLong

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

Example of user-defined date format

```
public class MyEntity {

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

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

Example of date stored as a long value

```
public class MyEntity {

    @DateLong
    private Date entityDate;
}
```

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

6.8.2. Custom Type Conversion

In order to define bespoke type conversions for particular members, you can annotate a field or method with `@Convert` to specify an implementation of `org.neo4j.ogm.typeconversion.AttributeConverter` to use.

Example of custom type converter

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

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

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

}
```

You could then apply this to your class as follows:

```
@NodeEntity
public class Invoice {

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

6.8.3. Spring's ConversionService

It is possible to have Spring Data Neo4j 4 use converters registered with Spring's ConversionService. In order to do this, provide `org.springframework.data.neo4j.conversion.MetaDataDrivenConversionService` as a Spring bean.

Provide MetaDataDrivenConversionService as a Spring bean

```
@Bean
public ConversionService conversionService() {
    return new MetaDataDrivenConversionService(getSessionFactory().metaData());
}
```

Then, instead of defining an implementation of `org.neo4j.ogm.typeconversion.AttributeConverter` on the `@Convert` annotation, use the `graphPropertyType` attribute to define the type to convert to.

Using graphPropertyType

```
@NodeEntity
public class MyEntity {

    @Convert(graphPropertyType = Integer.class)
    private DecimalCurrencyAmount fundValue;

}
```

Spring Data Neo4j 4 will look for converters registered with Spring's ConversionService that can convert both to and from the type specified by `graphPropertyType` and use them if they exist.

NOTE	Default converters and those defined explicitly via an implementation of `org.neo4j.ogm.typeconversion.AttributeConverter` will take precedence over converters registered with Spring's ConversionService.

6.8.4. Mapping Query Results

For queries executed via `@Query` repository methods, it's possible to specify a conversion of complex query results to POJOs. These result objects are then populated with the query result data and can be serialized and sent to a different part of the application, e.g. a frontend-ui. To take advantage of this feature, use a class annotated with `@QueryResult` as the method return type.

Example of query result mapping

```
public interface MovieRepository extends GraphRepository<Movie> {

    @Query("MATCH (movie:Movie)-[r:RATING]->(), (movie)<-[:ACTS_IN]-(actor:Actor) " +
           "WHERE movie.id={0} " +
           "RETURN movie as movie, COLLECT(actor) AS 'cast', AVG(r.stars) AS
'averageRating'")
    MovieData getMovieData(String movieId);

    @QueryResult
    public class MovieData {
        Movie movie;
        Double averageRating;
        Set<Actor> cast;
    }

}
```

6.9. Transactions

Neo4j is a transactional database, only allowing operations to be performed within transaction boundaries. Spring Data Neo4j integrates nicely with both the declarative transaction support with `@Transactional` as well as the manual transaction handling with `TransactionTemplate`. It also supports the rollback mechanisms of the Spring Testing library.

As of version 4.0, the classes used to perform transaction management have been rewritten. Instead of using `SpringTransactionManager` provided by the Neo4j kernel alongside Spring's `JtaTransactionManager`, the transaction management is performed by `Neo4jTransactionManager`, which implements Spring's `PlatformTransactionManager`.

This `Neo4jTransactionManager` is based on an OGM `Session`, on which the `beginTransaction()` method gets called, and this in turn delegates onto the underlying OGM's `TransactionManager` implementation.

The `Neo4jConfiguration` Spring configuration bean will create an instance of this `Neo4jTransactionManager` for use in Spring Data Neo4j. It is made available under the name "transactionManager" in the Spring application context.

6.10. Entity Attachment

In previous versions of Spring Data Neo4j, entities could be "attached" or "detached" depending on whether or not they were enhanced by AspectJ and actively managed by the framework. As of SDN 4, this is no longer the case and the AspectJ involvement has completely gone away.

6.10.1. Persisting Entities

From version 4 onwards, the entity persistence is all performed through the `save()` method on the

underlying `Session` object. This method is normally invoked indirectly via a Spring repository or `Neo4jTemplate`.

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

Example 1. Persisting entities

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

// Store Michael in the database.
Person p = new Person("Michael");
personRepository.save(p);
// or alternatively
neo4jTemplate.save(p);
```

6.10.2. 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. A depth of 0 will persist only the properties of the specified entity to the database, and a depth of -1 will persist everything in the object graph rooted at the given entity.

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

NOTE

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

Example 2. Relationship save cascading

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

@NodeEntity
class Actor {
    String name;
}

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

movie.setTopActor(actor);
```

Neither the actor nor the movie has been assigned a node in the graph. If we were to call `repository.save(movie)`, then Spring Data Neo4j 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 `repository.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.

If the relationships form a cycle, then the entities will first of all be assigned a node in the database, and then the relationships will be created. The cascading is however only propagated to related entity fields that have been modified.

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

Example 3. Cascade for modified fields

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

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

6.11. Sorting and Paging

Spring Data Neo4j supports sorting and paging of results, both when using Spring Data's `Pageable` and `Sort` interfaces, and also when using the `Session` object. The syntax for the two is slightly different, the main difference being that the `Session` object methods take independent arguments for sorting and pagination, whereas Spring's `Pageable` interface embeds an optional `Sort` object. Behind the scenes however, repository-based paging and sorting delegates to the core OGM equivalents.

NOTE Spring Data Neo4j 4 does not yet support sorting and paging on custom queries or derived query methods.

Repository-based paging

```
Pageable pageable = new PageRequest(0, 3);
Page<World> page = worldRepository.findAll(pageable, 0);
```

Repository-based sorting

```
Sort sort = new Sort(Sort.Direction.ASC, "name");
Iterable<World> worlds = worldRepository.findAll(sort, 0)) {
```

Repository-based sorting with paging

```
Pageable pageable = new PageRequest(0, 3, Sort.Direction.ASC, "name");
Page<World> page = worldRepository.findAll(pageable, 0);
```

Core OGM-based paging

```
Iterable<World> worlds = loadAll(World.class, new Pagination(0,3), 0)
```

Core OGM-based sorting

```
Iterable<World> worlds = loadAll(World.class, new SortOrder().add("name"), 0)
```

Core OGM-based sorting with paging

```
Iterable<World> worlds = loadAll(World.class, new SortOrder().add("name"), new
Pagination(0,3), 0)
```

6.12. Entity Type Representation

As of Spring Data Neo4j 4, type representation has been greatly simplified to the point that there is just one strategy. The `TypeRepresentationStrategy` has disappeared and a single label-based model is all that is supported.

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

Example domain model and labels

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

public class Person extends DomainObject {
   ...
}

public class Lady extends Person {
   ...
}

public class Gentleman extends Person {
   ...
}

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

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

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

In the current version, it is mandatory to specify the relationship type on a `@RelationshipEntity` annotation.

Chapter 7. Performance Considerations

As with any other object mapping framework, the domain entities that are created, read, or persisted potentially represent only a small fraction of the data stored in the database. This is the set needed for a certain use-case to be displayed, edited or processed in a low throughput fashion. The main advantages of using an object mapper in this case are the ease of use of real domain objects in your business logic and also the integration with existing frameworks and libraries that expect Java POJOs as input or create them as results.

Although adding layers of abstraction is a common pattern in software development, each of these layers generally add overhead and performance penalties. This chapter discusses the performance implications of using Spring Data Neo4j.

7.1. Focus on performance

This new version 4 of SDN has been rebuilt from the ground up. It is based on the understanding that the majority of users want to run application servers that connect to remote database instances. They will therefore need to communicate "over the wire". Neo4j provides the capability to do this now with its powerful Cypher language, which is exposed via a remote protocol.

What we have attempted to do is to ensure that, as much as possible, we don't overload that communication channel. This is important for two reasons. Firstly, every network interaction involves an overhead (both bandwidth but more so latency) which impacts the response times of your application. Secondly, network requests containing redundant operations (such as updaing an object which hasn't changed) are unnecessary, and have similar impacts. We have approached this problem in a number of ways:

7.1.1. 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, for example, you can set the depth to 2.

7.1.2. Smart object-mapping

SDN 4 introduces smart object-mapping. This means that, all other things being equal, it is possible to reliably detect which nodes and relationships need 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 updating, or create relationships that already exist. We can minimise the amount of data we send across the wire as a result, which leads to faster network interaction and fewer CPU cycles consumed on the server.

7.1.3. User-definable Session lifetime

Supporting the smart object-mapping capability is the `Neo4jSession`. This object can be declared with different lifetimes, depending on the requirements of your application. For web-based applications, you might choose between HTTP Request-scoped lifetime or HTTP Session-scoped lifetimes. For a standalone application, you may choose to maintain a single session for the entire lifetime of the application.

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.

It is also possible to manage your session lifetimes in code. For example, associated with single *fetch-update-save* cycle or unit of work.

Migration Guide

Chapter 8. Migrating from previous versions of Spring Data Neo4j

8.1. Package Changes

Because the Neo4j Object Graph Mapper can be used independently of Spring Data Neo4j, the core annotations have been moved out of the spring framework packages:

`org.springframework.data.neo4j.annotation` → `org.neo4j.ogm.annotation`

NOTE	The `@Query` and `@QueryResult` annotations are only supported in the Spring modules, and are not used by the core mapping framework. These annotations have not changed.

8.2. Annotation Changes

There have been some changes to the annotations that were used in previous versions of Spring Data Neo4j. Wherever possible we have tried to maintain the previous annotations verbatim, but in a few cases this has not been possible, usually for technical reasons but sometimes for aesthetic ones. Our goal has been to minimise the number of annotations you need to use as well as trying to make them more self-explanatory. The following annotations have been changed.

Old	New
`@RelatedTo`	`@Relationship`
`@RelatedToVia`	`@Relationship`
`@GraphProperty`	`@Property`
`Relationship Direction.BOTH`	`Relationship.UNDIRECTED`

8.3. Custom Type Conversion

SDN 4 provides automatic type conversion for the obvious candidates: byte[] and Byte[] arrays, Dates, BigDecimal and BigInteger types. In order to define bespoke type conversions for particular entity attribute, you can annotate a field or method with `@Convert` to specify your own implementation of `org.neo4j.ogm.typeconversion.AttributeConverter`.

You can find out more about type conversions here: Custom Converters

8.4. Date Format Changes

The default Date converter is @DateString.

SDN 3.x and earlier represented Dates as a String value consisting of the number of milliseconds since January 1, 1970, 00:00:00 GMT.

If you are upgrading to SDN 4.x from these versions and your application used the default, then you need to annotate your `Date` properties with `@DateLong`. Moreover, the property values in the graph need to be converted to numbers.

Upgrade Date properties to numbers

```
MATCH (n:Foo) //All nodes which contain date properties to be migrated
WHERE NOT HAS(n.migrated)// Take the first 10k nodes that haven't been migrated yet
WITH n LIMIT 10000
SET n.dateProperty = toInt(n.dateProperty),n.migrated=1 //where dateProperty is the
date with a String value to be migrated
RETURN count(n); //Run until the statement returns zero records
//Similar process to remove the migrated flag
```

However, if your application already represented Dates as `@GraphProperty(propertyType = Long.class)` then simply changing this to `@DateLong` is sufficient.

8.5. Obsolete Annotations

The following annotations are no longer used, either because they are no longer needed, or cannot be supported via Cypher.

- @GraphTraversal
- @RelatedToVia
- @RelatedTo
- @Index
- @TypeAlias
- @Fetch

8.6. Features No Longer Supported

Some features of the previous annotations have been dropped.

8.6.1. Overriding @Property Types

Support for overriding property types via arguments to @Property has been dropped. If your attribute requires a non-default conversion to and from a database property, you can use a Custom Converter instead.

8.6.2. @Relationship enforceTargetType

In previous versions of Spring Data Neo4j, you would have to add an `enforceTargetType` attribute into every clashing `@Relationship` annotation. Thanks to changes in the underlying object-graph mapping mechanism, this is no longer necessary.

Clashing Relationship Types

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

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

8.6.3. Cross-store Persistence

Neo4j is dropping XA support and therefore SDN 4 does not provide any capability for cross-store persistence

8.6.4. TypeRepresentationStrategy

SDN 4 replaces the existing TypeRepresentionStrategy configuration with a straightforwad convention based on simple class-names or entities using `@NodeEntity(label=⋯)`

Please refer to Entity Type Representation for more details.

8.6.5. AspectJ Support

Support for AspectJ-based persistence has been removed from SDN 4 as the write-and-read-through approach only works with an integrated, embedded database, not Neo4j server. The performance improvements in SDN 4 should make their use as a performance optimisation unnecessary anyway.

8.7. Changes to Neo4jTemplate

The `Neo4jTemplate` has been slimmed-down significantly for SDN 4 with many of the method signatures changed to reflect the updated behaviour.

Many of the operations are no longer needed or can be expressed with a straightforward Cypher query.

Developers are also encouraged to code against its `Neo4jOperations` interface instead of the template class.

8.7.1. API Changes

The following table shows the `Neo4jTemplate` functions that have been retained for version 4 of Spring Data Neo4j. In some cases the method names have changed but the same functionality is offered under the new version.

Table 1. Neo4j Template Method Migration

Old Method Name	New Method Name	Notes
`findOne`	`load`	Overloaded to take optional depth parameter
`findAll`	`loadAll`	Overloaded to take optional depth parameter, also now returns a `Collection` rather than a `Result`
`query`	`query`	Return type changed from `Result` to be `Iterable`
`save`	`save`	
`delete`	`delete`	
`count`	`count`	No longer defines generic type parameters
`findByIndexedValue`	`loadByProperty`	Indexes are not supported natively, but you can index node properties in your database setup and use this method to find by them

To achieve the old `template.fetch(entity)` equivalent behaviour, you should call one of the load methods specifying the fetch depth as a parameter.

It's also worth noting that `exec(GraphCallback)` and the `create…()` methods have been made obsolete by Cypher. Instead, you should now issue a Cypher query to the new `execute` method to create the nodes or relationships that you need.

Dynamic labels, properties and relationship types are not supported as of this version, server extensions should be considered instead.

8.8. Indexing

The best way to retrieve start nodes for traversals and queries is by using Neo4j's integrated index facilities. Spring Data Neo4j takes the view that index maintenance should not be part of your application code. For that reason, it does not provide any explicit index-related functionality.

However, it is very important that indices are in place for efficient node lookups. Make sure those are applied to your test and especially production databases to guarantee efficient query execution. Please make sure you read the Neo4j Documentation on indices.

8.8.1. Built-In Query DSL Support

Previous versions of SDN allowed you to use a DSL to generate Cypher queries. There are many different DSL libraries available and you're free to use which of these - or none - that you want. With Cypher changing on a regular basis, avoiding a DSL implementation in SDN means less ongoing maintenance and less likelihood of your code being incompatible with future versions of Neo4j.

8.8.2. Graph Traversal and Node/Relationship Manipulation

These features cannot be supported by Cypher and have therefore been dropped from `Neo4jTemplate`.

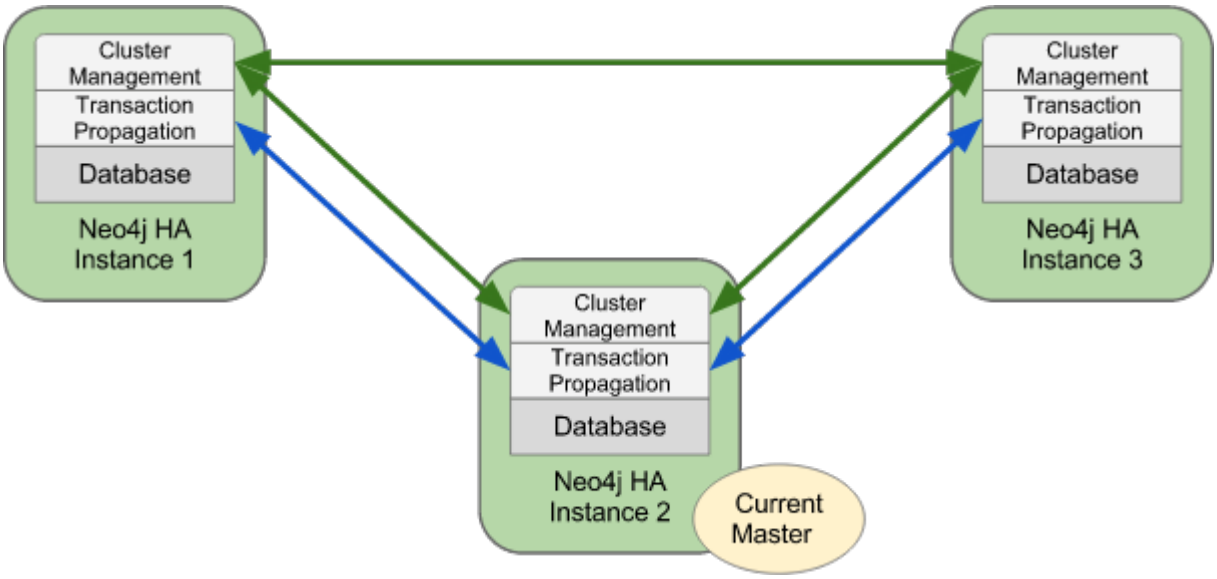
Please provide feedback on the new APIs of SDN 4 and the migration needs to spring-data-neo4j@neotechnology.com or via a JIRA issue

Configuration in an HA Environment

Chapter 9. Configuring Spring Data Neo4j 4.1 in an HA Environment

9.1. Transaction Binding in HA Mode

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



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

9.2. Read-only Transactions

As of version 4, Spring Data Neo4j does not distinguish between WRITE transactions and READ-ONLY transactions. We cannot therefore bind read-only transactions to slaves and write transactions to master. A future version will address this deficiency, but in the meantime the only way to ensure that everything works as expected is to direct every transaction to master. There are a couple of ways to to achieve this.

9.3. Static Binding to a Designated Master

9.3.1. Example cluster:

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

SDN4 Binding to master IP address

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

NOTE

We don't really recommend this approach, except for testing purposes and non-critical deployments. Firstly, it will only work if you always bring up the designated master first, and secondly, if the master goes down all subsequent transactions will fail until it is restarted. In HA mode, the cluster is able to elect a new master when this happens, but as of version 4 of Spring Data Neo4j, there is no mechanism for querying the cluster to identify the current master. The solution in this case is to use a load balancer such as HAProxy that can do this for us. This is described in the next section.

9.4. Dynamic Binding via a Load Balancer

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

9.4.1. 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

Spring Data Neo4j 4 Binding via HAProxy

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

Sample haproxy.cfg

```
global
    daemon
    maxconn 256

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

frontend http-in
    bind *:80
    default_backend neo4j

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

listen admin
    bind *:8080
    stats enable
```

Appendix

Chapter 10. Appendix

10.1. Repository Query Keywords

The following table lists the keywords generally supported by the Spring Data Neo4j repository query derivation mechanism.

Table 2. Query Keywords

Logical keyword	Keyword expressions	Restrictions
`AND`	`and`	
`OR`	`or`	Cannot be used to OR nested properties
`GREATER_THAN`	`GreaterThan`	
`LESS_THAN`	`LessThan`	
`LIKE`	`Like`, `IsLike`	
`NOT`	`Not`	
`NOT_LIKE`	`NotLike`, `IsNotLike`	
`REGEX`	`Matches`	