Good Relationships

The Spring Data Neo4j Guide Book

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

Version 5.0.0.M1, 2016-11-23

Table of Contents

Spring Data Neo4j Version 4.2 (July 2016)

Foreword

I'm excited about Spring Data Neo4j for several reasons.

First, this project is in a very important space. We are in an era of transition. A very few years ago, a relational database was a given for storing nearly all the data in nearly all applications. While relational databases remain important, new application requirements and massive data proliferation have prompted a richer choice of data stores. Graph databases have some very interesting strengths, and Neo4j is proving itself valuable in many applications. It's a choice you should add to your toolbox.

Second, Spring Data Neo4j is an innovative project, which makes it easy to work with one of the most interesting new data stores. Unfortunately, the proliferation of new data stores has not been matched by innovation in programming models to work with them. Ironically, just after modern ORM mapping made working with relational data in Java relatively easy, the data store disruption occurred, and developers were back to square one: struggling once more with clumsy, low level APIs. Working with most non-relational technologies is overly complex and imposes too much work on developers. Spring Data Neo4j makes working with Neo4j amazingly easy, and therefore has the potential to make you more successful as a developer. Its use of AspectJ to eliminate persistence code from your domain model is truly innovative, and on the cutting edge of today's Java technologies.

Third, I'm excited about Spring Data Neo4j for personal reasons. I no longer get to write code as often as I would like. My initial convictions that Spring and AspectJ could both make building applications with Neo4j dramatically easier and cross-store object navigation possible gave me an excuse for a much-needed coding binge early in 2010. This led to a prototype of what became Spring Data Neo4j — at times written paired with Emil. I'm sure the vast majority of my code has long since been replaced (probably for the better) by coders who aren't rusty — thanks Michael and Thomas! — but I retain my pleasant memories.

Finally, Spring Data Neo4j is part of the broader Spring Data project: one of the key areas in which Spring is innovating to help meet new application requirements. I encourage you to explore Spring Data, and — better still — become involved in the community and contribute.

Enjoy the Spring Data Neo4j book, and happy coding!

— Rod Johnson, Founder of the Spring Framework

> "Spring is the most popular middleware on the planet," I thought to myself as I walked up to Rod Johnson in late 2009 at the JAOO conference in Aarhus, Denmark. Rod had just given an introductory presentation about Spring Roo and when he was done I told him "Great talk. You're clearly building a stack for the future. What about support for non-relational databases?"
>
> We started talking and quickly agreed that NOSQL will play an important role in emerging stacks. Now, a year and half later, Spring Data Neo4j is available in its first stable release and I'm blown away by the result. Never before in any environment, in any programming framework, in any stack, has it been so easy and intuitive to tap into the power of a graph database like Neo4j. It's a testament to the efforts by an awesome team of four hackers from Neo Technology and VMware: Michael Hunger, David Montag, Thomas Risberg and Mark Pollack.
>
> The Spring framework revolutionized how we all wrote enterprise Java applications and today it's used by millions of enterprise developers. Graph databases also stand out in the NOSQL crowd when it comes to enterprise adoption. You can find graph databases used in areas as diverse as network management, fraud detection, cloud management, anything with social data, geo and location services, master data management, bioinformatics, configuration databases, and much more.
>
> Spring developers deserve access to the best tools available to solve their problem. Sometimes that's a relational database accessed through JPA. But more often than not, a graph database like Neo4j is the perfect fit for your project. I hope that Spring Data Neo4j will give you access to the power and flexibility of graph databases while retaining the familiar productivity and convenience of the Spring framework.
>
> Enjoy the Spring Data Neo4j guide book and welcome to the wonderful world of graph databases!
>
> — Emil Eifrem, CEO of Neo Technology

Preface

Welcome to the Spring Data Neo4j Guide Book.

This 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.

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.

It is written by developers for developers. Enjoy the book!

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.

Introduction

Chapter 1. Spring Data

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 (SDN for short) project, as part of the Spring Data initiative, aims to simplify development with the Neo4j graph database. Like JPA, SDN uses annotations on simple POJO domain objects which it then stores as metadata which in turn helps it to drive mapping into entities and their fields to nodes, relationships, and properties into and from the graph database.

Chapter 2. 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 been used 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 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 4.2 onwards) provides facilities for handling that setup out of the box.

During development it can be handy to define Indexes that change as your domain model changes. However you do not want this to be the case in all environments. SDN provides a validation option that makes sure your indexes are present on startup.

Note that these facilities are only available on Neo4j 3+.

For more information about Indexes and Constrains in Neo4j see the Neo4j Reference Manual.

Reference Documentation

This reference guide covers information about the programming model, APIs, concepts, annotations and technical details of Spring Data Neo4j Version 4.2.

Chapter 3. Overview

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 introduced support for connecting to an embedded instance of Neo4j and connecting to a remote instance using the Bolt protocol introduced by Neo4j 3.0. Version 4.2 introduces an updated developer API more akin to other Spring Data Projects.

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

The explanation of Spring Data Neo4j's programming model starts with some underlying details. The basic concepts of the http://neo4j.com/docs/ogm-manual/current/ (OGM) library] used by Spring Data Neo4j internally, is explained in the initial chapter.

3.1. Features

Below you'll find a quick run down of the main features SDN provides.

3.1.1. 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 Neo4j.

3.1.2. 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.

3.1.3. 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).

3.1.4. Sessions

Spring Data Neo4j supports the usage of OGM Sessions for interacting with the mapped entities and the Neo4j graph database if you don't want to use repositories. Support for Spring Data Neo4j Repositories is also based on the `Session`, so the underlying functionality is identical.

3.1.5. 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.

3.1.6. 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.

3.1.7. Configuration

Currently, only JavaConfig configuration is supported. XML based configuration has experimental support. See Spring configuration for more details.

3.1.8. Performance

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

3.2. Getting Help

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

To learn more refer to:

- the Reference Manual;
- the sample project: SDN Univeristy. More example projects for Spring Data Neo4j 4 are available in the Neo4j-Examples repository
- The main SpringSource project site contains links to basic project information such as source code, JavaDocs, Issue tracking, etc.
- the Javadocs;
- for more detailed questions, use Spring Data Neo4j on StackOverflow

If you are new to Spring as well as to Spring Data, look for information about Spring projects.

3.3. Got feedback?

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 document. Hopefully we've created a guide that is well-received by our peers in the development community but sometimes things aren't detailed enough or have enough documentation. We are a very quick turnaround development team so if you don't see what, please inform us about missing or incorrect content.

If you have any feedback on Spring Data Neo4j or this book, please provide it via:

- SpringSource JIRA
- StackOverflow
- The Neo4j Google Group.

Chapter 4. Requirements

Spring Data Neo4j 4.2.x at minimum, requires:

- JDK Version 8 and above.
- Neo4j Database 2.3.x and above.
- Spring Framework 4.3.x and above.

Chapter 5. Spring Data Release Train Dependencies

Due to different inception dates of individual Spring Data modules, most of them carry different major and minor version numbers. The easiest way to find compatible ones is by relying on the Spring Data Release Train BOM we ship with the compatible versions defined. In a Maven project you'd declare this dependency in the `<dependencyManagement />` section of your POM:

Example 1. Using the Spring Data release train BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>${release-train}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `Ingalls-M1`. The train names are ascending alphabetically and currently available ones are listed here. The version name follows the following pattern: `${name}-${release}` where release can be one of the following:

- `BUILD-SNAPSHOT` - current snapshots
- `M1`, `M2` etc. - milestones
- `RC1`, `RC2` etc. - release candidates
- `RELEASE` - GA release
- `SR1`, `SR2` etc. - service releases

A working example of using the BOMs can be found in our Spring Data examples repository. If that's in place declare the Spring Data modules you'd like to use without a version in the `<dependencies />` block.

Example 2. Declaring a dependency to a Spring Data module

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j</artifactId>
  </dependency>
<dependencies>
```

5.1. Dependency management with Spring Boot

Spring Boot already selects a very recent version of Spring Data modules for you. In case you want to upgrade to a newer version nonetheless, simply configure the property `spring-data-releasetrain.version` to the train name and iteration you'd like to use.

To override the version of the OGM that you want to use configure the property `neo4j-ogm.version`.

5.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.0.0.M3 or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

Chapter 6. Architecture

While SDN 4 has been built from the ground up it is important to understand a little about it's architecture as it has significantly diverged from SDN 3 and could have implications in how you design your application.

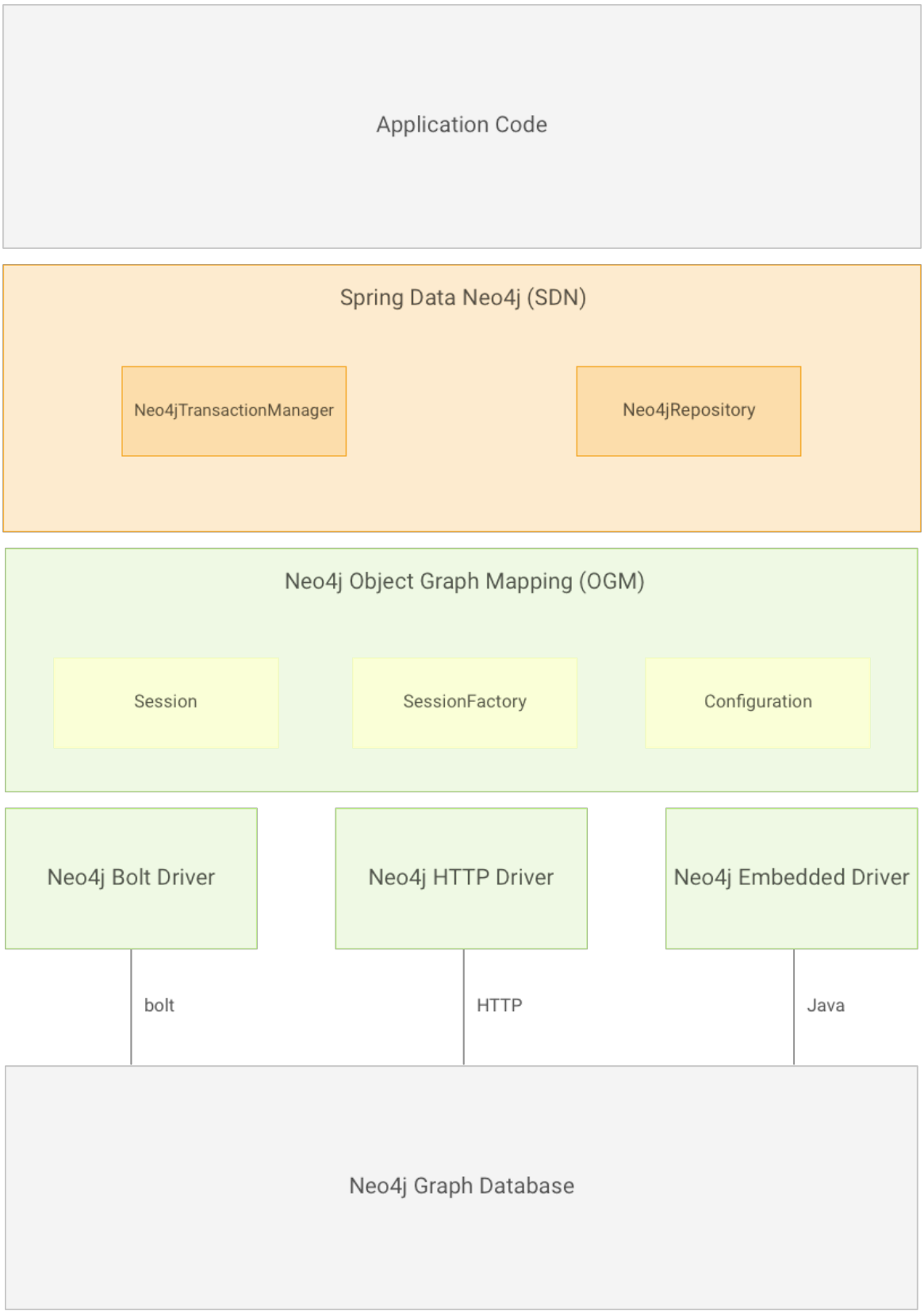
6.1. Overview

SDN 3.x was monolithic in a sense that all mapping code, database drivers and Spring integrations were all in one. SDN 4 has decided to break these components up into:

- Drivers: At the moment these come in 3 variants: Embedded, HTTP and the binary protocol Bolt.
- The Object Graph Mapper (OGM): This is similar to an ORM in that it maps database nodes to java objects. This library is agnostic of any framework (including Spring).
- Spring Data Neo4j 4: Provides syntactic sugar and code on top of the OGM to help quickly build Spring Based Neo4j OGM apps.

Those coming from other Spring Data projects or are familiar with ORM products like JPA or Hibernate may quickly recognise this architecture. A bulk of the heavy lifting has been moved into the OGM. The OGM's key interfaces that you will deal with regularly are the `Session` and `SessionFactory`. It is worth understanding a little more about them.

The following diagram outlines how the components defined above:


Application Code
Spring Data Neo4j (SDN)
Neo4jTransactionManager
Neo4jRepository
Neo4j Object Graph Mapping (OGM)
Session
SessionFactory
Configuration
Neo4j Bolt Driver
Neo4j HTTP Driver
Neo4j Embedded Driver
bolt
HTTP
Java
Neo4j Graph Database

6.2. Neo4j OGM

6.2.1. SessionFactory

The `SessionFactory` is needed by SDN 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.

6.2.2. Session

A `Session` is used to drive the object-graph mapping framework. All repository implementations are driven by the `Session`. 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.

For most request/response type applications SDN will take care of Session management for you (as defined in the Configuration section above). If you have a batch or long running desktop type application you may want to know how you can control using the session a bit more.

Design Consideration: Session caching

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 `org.neo4j.ogm.session.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 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.

Chapter 7. 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.

Examples for these different setups can be found in the Spring Data Neo4j examples.

7.1. Dependency Management

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.

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

7.1.1. Maven

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

By default, SDN 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 or Bolt drivers in your production application, you must add the following dependencies as well. (This dependency on the embedded driver 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>

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

If you'd rather like the latest snapshots of the upcoming major version, use our Maven snapshot repository and declare the appropriate dependency version.

```
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-neo4j</artifactId>
        <version>4.2.X.BUILD-SNAPSHOT</version>
    </dependency>

    <!-- used for nightly builds -->
    <repository>
      <id>spring-maven-snapshot</id>
      <snapshots><enabled>true</enabled></snapshots>
      <name>Springframework Maven SNAPSHOT Repository</name>
      <url>http://repo.spring.io/libs-release</url>
    </repository>

    <!-- used for milestone/rc releases -->
    <repository>
      <id>spring-maven-milestone</id>
      <name>Springframework Maven Milestone Repository</name>
      <url>http://repo.spring.io/libs-milestone</url>
    </repository>
```

Testing

Maven dependencies for testing SDN 4 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

Since 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.

7.1.2. Gradle

Gradle dependencies are basically the same as Maven:

```
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}'

    # add this dependency if you want to use the Bolt driver
    compile 'org.neo4j:neo4j-ogm-bolt-driver:{ogm-version}'

}
```

Similar to maven, if you want to use either the latest snapshots or milestones, you will need to add the following:

```
repositories {
   # used for milestone/rc releases
   maven { url "http://repo.spring.io/libs-milestone" }

   # used for nightly builds
   maven { url "http://repo.spring.io/libs-snapshot" }
}
```

7.1.3. Ivy

Ivy dependencies are also similar to Maven:

```
<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}"/>

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

7.2. Spring configuration

Version 4.2 significantly reduces the complexity of configuration for the application developer.

Right now SDN only supports JavaConfig. There is no XML based support at this stage.

NOTE	If you have come from Version 4.1 or earlier you no longer need to define a `Session` bean and Spring `Scope` or extend from `Neo4jConfiguration`. This is now all taken care for you by SDN.

For most applications the following configuration is all that's needed to get up and running.

7.2.1. Basic Java Spring Applications

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

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

    @Bean
    public Neo4jTransactionManager transactionManager() {
        return new Neo4jTransactionManager(sessionFactory());
    }

}
```

7.2.2. Spring WebMVC Applications

If you are using a Spring WebMVC application, the following configuration is all that's required:

```
@Configuration
@EnableWebMvc
@ComponentScan({"org.neo4j.example.web"})
@EnableNeo4jRepositories("org.neo4j.example.repository")
@EnableTransactionManagement
public class MyWebAppConfiguration extends WebMvcConfigurerAdapter {

    @Bean
    public OpenSessionInViewInterceptor openSessionInViewInterceptor() {
        OpenSessionInViewInterceptor openSessionInViewInterceptor =
            new OpenSessionInViewInterceptor();
        openSessionInViewInterceptor.setSessionFactory(sessionFactory());
        return openSessionInViewInterceptor;
    }

    @Override
        public void addInterceptors(InterceptorRegistry registry) {
        registry.addWebRequestInterceptor(openSessionInViewInterceptor());
    }

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

    @Bean
    public Neo4jTransactionManager transactionManager() throws Exception {
        return new Neo4jTransactionManager(sessionFactory());
    }
}
```

7.2.3. Java Servlet Container Applications

If you are using a Java Servlet 3.x+ Container, you can configure a Servlet filter with Spring's `AbstractAnnotationConfigDispatcherServletInitializer`. The configuration below will open a new session for every web request then automatically close it on completion. SDN provides the `org.springframework.data.neo4j.web.support.OpenSessionInViewFilter` to do this:

```
public class MyAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected void customizeRegistration(ServletRegistration.Dynamic registration) {
      registration.setInitParameter("throwExceptionIfNoHandlerFound", "true");
  }

  @Override
  protected Class<?>[] getRootConfigClasses() {
      return new Class[] {ApplicationConfiguration.class} // if you have broken up
your configuration, this points to your non web application config/s.
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
      throw new Class[] {WebConfiguration.class}; // a configuration that extends the
WebMvcConfigurerAdapter as seen above.
  }

  @Override
  protected String[] getServletMappings() {
    return new String[] {"/"};
  }

  protected Filter[] getServletFilters() {
    return return new Filter[] {new OpenSessionInViewFilter()};
  }

}
```

7.2.4. Spring Boot Applications

Unfortunately, due to the Spring release schedule you cannot use the current or upcoming versions of Spring Boot and expect to use these new improvements out of the box.

That said Spring Boot is flexible enough for you to override the current configuration and supply it with something compatible with SDN 4.2.

To do that update the Spring Boot properties to use the current SDN version. Update your Spring Boot Maven POM with the following. You may need to add `<repositories>` depending on versioning.

```
    ...
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <spring-data-releasetrain.version>Ingalls-BUILD-SNAPSHOT</spring-data-
releasetrain.version>
        <neo4j-ogm.version>2.1.0-SNAPSHOT</neo4j-ogm.version>
    </properties>
    ...
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-neo4j</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
    </dependencies>
    ...
```

Remove the reference to the old Neo4j Configuration where you define your Spring Boot application using the `exclude` option in the `@SpringBootApplication`.

```
@Controller
@SpringBootApplication(exclude={Neo4jDataAutoConfiguration.class})
public class Application {
    public static main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Finally, add the your new SDN 4.2 Configuration somewhere in the Boot classpath (it will pick up your configuration if it's annotated with `@Configuration`:

```
@Configuration
@EnableNeo4jRepositories(basePackages = "your.project.repository")
@EnableTransactionManagement
@ComponentScan({"your.project.services"})
@EnableWebMvc
@EnableConfigurationProperties(Neo4jProperties.class)
public class Neo4jConfiguration extends WebMvcConfigurerAdapter {

    private final Neo4jProperties properties;

    public Neo4jConfiguration(Neo4jProperties properties) {
        this.properties = properties;
    }

    @Bean
    @ConditionalOnMissingBean
    public org.neo4j.ogm.config.Configuration configuration() {
        return this.properties.createConfiguration();
    }

    @Bean
    public OpenSessionInViewInterceptor openSessionInViewInterceptor() {
        OpenSessionInViewInterceptor openSessionInViewInterceptor =
                new OpenSessionInViewInterceptor();
        openSessionInViewInterceptor.setSessionFactory(sessionFactory(configuration()
));
        return openSessionInViewInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addWebRequestInterceptor(openSessionInViewInterceptor());
    }

    @Bean
    public SessionFactory sessionFactory(org.neo4j.ogm.config.Configuration
configuration) {
        return new SessionFactory(configuration, "your.project.domain");
    }

    @Bean
    public Neo4jTransactionManager transactionManager(SessionFactory sessionFactory) {
        return new Neo4jTransactionManager(sessionFactory);
    }
}
```

7.3. Driver Configuration

SDN 4 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
- Bolt driver

By default, SDN 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 using both techniques.

7.3.1. 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 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 configuration() {
   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 sessionFactory() {
    return new SessionFactory(configuration(), <packages> );
}
```

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

7.3.2. Bolt Driver

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

ogm.properties

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

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

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

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

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

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

Java Configuration

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


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

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

7.3.3. 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 configuration() {
   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 sessionFactory() {
    return new SessionFactory(configuration(), <packages> );
}
```

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

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

7.3.4. Authentication

If you are using the Http or Bolt 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 configuration() {
   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 cnfiguration() {
   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 cnfiguration() {
   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`

7.4. Transport Layer Security

The Bolt and Http drivers allow you to connect to Neo4j over a secure channel. These rely on Transport Layer Security (aka 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.

ogm.properties (bolt):

```
encryption.level=REQUIRED
trust.strategy=TRUST_ON_FIRST_USE
trust.certificate.file=/tmp/cert
```

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.

ogm.properties (driver):

```
trust.trategy = ACCEPT_UNSIGNED
```

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

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

Chapter 8. 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.

8.1. Under the hood

8.1.1. 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`.

8.1.2. The Session object

The Spring repositories are 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. SDN 4.2 allows users to `@Autowire Session` into Spring managed services directly to get access to a contextual Session bean in that current Thread's scope.

8.1.3. 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.

8.1.4. 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.

NOTE

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 `org.neo4j.ogm.session.Session.clear()`.

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.

8.2. 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 `Session` 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.

8.3. 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.

8.3.1. @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.

8.3.2. @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!

WARNING	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 SDN populates the database ID on save. This causes problems if you had inserted the newly created entity into a hash-based collection before saving.

8.3.3. @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`.

8.3.4. Runtime Managed Labels

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.

8.4. 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.

8.4.1. @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.

8.4.2. @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.

8.4.3. 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.

8.4.4. 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.

8.5. 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.

8.5.1. Index Management in Spring Data Neo4j 4

SDN 4.2 allows developers to mark up OGM managed classes with the `@Index` annotation.

To mark a field as indexed simply add the `@Index` annotation. If your field must be unique add `@Index(unique=true)`. 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.

By default index management is set to `None`.

Once you have annotated your classes you then have various Auto Indexing Options. SDN will try to configure the the auto indexer from the `ogm.properties` file, which it expects to find on the classpath or via Spring through a `Configuration` bean.

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

ogm.properties

```
indexes.auto=assert
```

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:

```
@Bean
public Configuration configuration() {
   Configuration config = new Configuration();
   config
       .autoIndexConfiguration()
       .setAutoIndex("assert");
   return config;
}

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

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	indexes.auto=validate	config.autoIndexConfiguration().setAutoIndex("validate");
assert	Drops all constraints and indexes on startup then builds indexes based on whatever is represented in OGM by `@Index`. Handy during development	indexes.auto=assert	config.autoIndexConfiguration().setAutoIndex("assert");
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.	indexes.auto=dump indexes.auto.dump.dir=<a directory> indexes.auto.dump.filename=<a filename>	config.autoIndexConfiguration().setAutoIndex("dump"); config.autoIndexConfiguration().setDumpDir("XXX"); config.autoIndexConfiguration().setDumpFilename("XXX");

8.5.2. Index queries in Repositories

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

8.5.3. Legacy 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.

8.5.4. 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.

NOTE

These methods work only with the HTTP and Embedded drivers. The Bolt driver does not support updates to legacy indexes. If any of the methods below are employed, the user is responsible for managing resources such as the underlying GraphDatabaseService if the Driver used to get hold of the underlying implementation.

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 as a `@Query` defined in a repository class.

8.5.5. 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.

8.6. Using the OGM Session

WARNING	`Session` now replaces `Neo4jTemplate` functionality as all functionality can be found on the OGM `Session` object.

SDN now allows you to wire up the OGM `Session` directly into your Spring managed beans.

While SDN `Repository` will cover a majority of user scenarios sometimes it doesn't offer enough options. The OGM's `Session` offers a convenient API to interact more tightly with a Neo4j graph database.

8.6.1. Basic Operations

For Spring Data Neo4j 4, low level operations are handled by the OGM `Session`. 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.

8.6.2. Entity Persistence

`Session` allows you to `save`, `load`, `loadAll` and `delete` entities. 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 by SDN's Transaction Manager bean.

8.6.3. Cypher Queries

The `Session` also allows execution of arbitrary Cypher queries via its `query`, `queryForObject` and `queryForObjects` methods. Cypher queries that return tabular results should be passed into the `query` method. An `org.neo4j.ogm.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, of which nodes and relationships are mapped to domain entities if possible. 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.

8.6.4. Transactions

If you configured the `Neo4jTransactionManager` bean, any `Session` that is managed by Spring will automatically take part in Thread contextual Transactions. In order to do this you will need to wrap your service code using `@Transactional` or the `TransactionTemplate`.

NOTE	It is important to know that if you enable Transactions **ALL** code that uses the `Session` or a `Repository` must be enclosed in a @Transactional annotation.

For more details see Transactions

8.7. 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 `Neo4jRepository<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	`Neo4jRepository` 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.

8.7.1. Neo4jRepository

As of SDN 4, this `Neo4jRepository<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 `Neo4jRepository` 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()`

8.7.2. GraphRepository (Version 4.0.x - 4.1.x)

`Neo4jRepository` has replaced `GraphRepository` but essentially have the same features. This is only provided for legacy reasons and has been deprecated.

8.7.3. 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. All other parameters types are provided directly (i.e. Strings, Longs, etc).

There is special support for the Pageable parameter from Spring Data Commons, which is supported to add programmatic paging and slicing(alternatively static paging and sorting can be supplied in the query string itself).

If it is required that paged results return the correct total count, the @Query annotation can be supplied with a count query in the countQuery attribute. This query is executed separately after the result query and its result is used to populate the number of elements on the Page.

NOTE	Custom queries do not support a custom depth. Additionally, `@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 Neo4jRepository<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 a Page of Actors that have a ACTS_IN relationship to the movie node
with the title equal to movieTitle parameter.
    @Query("MATCH (movie:Movie {title={0}})<-[:ACTS_IN]-(actor) RETURN actor")
    Page<Actor> getActorsThatActInMovieFromTitle(String movieTitle, PageRequest page);

    // returns a Page of Actors that have a ACTS_IN relationship to the movie node
with the title equal to movieTitle parameter with an accurate total count
    @Query("MATCH (movie:Movie {title={0}})<-[:ACTS_IN]-(actor) RETURN actor",
countQuery = "MATCH (movie:Movie {title={0}})<-[:ACTS_IN]-(actor) RETURN count(*)")
    Page<Actor> getActorsThatActInMovieFromTitle(String movieTitle, Pageable page);

    // returns a Slice of Actors that have a ACTS_IN relationship to the movie node
with the title equal to movieTitle parameter.
    @Query("MATCH (movie:Movie {title={0}})<-[:ACTS_IN]-(actor) RETURN actor")
    Slice<Actor> getActorsThatActInMovieFromTitle(String movieTitle, Pageable 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 Neo4jRepository<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);

    // MATCH (person:Person)
    // WHERE person.age = {0}
    // RETURN person ORDER BY person.name SKIP {skip} LIMIT {limit}
    Page<Person> findByAge(int age, Pageable pageable);

    // MATCH (person:Person)
    // WHERE person.age = {0}
    // RETURN person ORDER BY person.name
    List<Person> findByAge(int age, Sort sort);

    //Allow a custom depth as a parameter
    Person findByName(String name, @Depth int depth);

    //Fix the depth for the query
    @Depth(value = 0)
    Person findBySurname(String surname);

}
```

8.7.4. Creating repositories

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

Using basic Neo4jRepository CRUD-methods

```
@Repository
public interface PersonRepository extends Neo4jRepository<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.

8.8. 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.

8.8.1. 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.

8.8.2. 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;
   ...
}
```

8.8.3. 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.

8.8.4. 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;
    }

}
```

8.9. 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`.

Demarcating @Transactional is required for all methods that interact with SDN. CRUD methods on `Repository` instances are transactional by default. If you are simply just looking up an object through a repository for example, then you do not need to define anything else: SDN will take of everything for you. That said, it is strongly recommended that you always annotate any service boundaries to the database with a `@Transactional` annotation. This way all your code for that method will always run in one transaction, even if you add a write operation later on.

More standard behaviour with Transactions is using a facade or service implementation that typically covers more than one repository or database call as part of a 'Unit of Work'. Its purpose is to define transactional boundaries for non-CRUD operations:

IMPORTANT	It is important to know that if you enable Transactions **ALL** code that uses the `Session` or `Neo4jTemplate` must have a @Transactional annotation.
NOTE	SDN only supports PROPAGATION_REQUIRED and ISOLATION_DEFAULT type transactions.

Using a facade to define transactions for multiple repository calls

```
@Service
class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  @Autowired
  public UserManagementImpl(UserRepository userRepository,
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```

This will cause call to `addRoleToAllUsers(…)` to run inside a transaction (participating in an existing one or create a new one if none already running). The transaction configuration at the repositories will be neglected then as the outer transaction configuration determines the actual one used.

It is highly recommended that users understand how Spring Transactions work. Below are some excellent resources:

- Spring Transaction Management
- Upgrading to Spring Data Neo4j 4.2

Read only Transactions

As of SDN 4.2 you can also define read only transactions.

You can start a read only transaction by marking a class or method with `@Transactional(readOnly=true)`.

CAUTION	Note that if you open a read only transaction from, for example a service method, and then call a mutating method that is marked as read/write your transaction semantics will always be defined by the outermost transaction. Be wary!

Configuration

Note that you will have to activate `@EnableTransactionManagement` explicitly to get annotation based configuration at facades working as well as define an instance of this `Neo4jTransactionManager` with the bean name `transactionManager`. The example above assumes you are using component scanning.

To allow your query methods to be transactional simply use @Transactional at the repository interface you define.

8.9.3. Transaction Bound Events

Formally Data Manipulation Events/ Lifecycle Events

SDN provides the ability to bind the listener of an event to a phase of the transaction. The typical example is to handle the event when the transaction has completed successfully: this allows events to be used with more flexibility when the outcome of the current transaction actually matters to the listener.

Spring Framework is currently structured in such a way that the context is not aware of the transaction support and has an open infrastructure to allow additional components to be registered and influence the way event listeners are created.

The transaction module implements an `EventListenerFactory` that looks for the new `@TransactionalEventListener` (as of Spring 4.2) annotation. When this one is present, an extended event listener that is aware of the transaction is registered instead of the default.

Example: An order creation listener.

```
@Component
public class MyComponent {

  @TransactionalEventListener(condition = "#creationEvent.awesome")
  public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {
    ...
  }

}
```

`@TransactionalEventListener` is a regular `@EventListener` and also exposes a `TransactionPhase`, the default being `AFTER_COMMIT`. You can also hook other phases of the transaction (`BEFORE_COMMIT`, `AFTER_ROLLBACK` and `AFTER_COMPLETION` that is just an alias for `AFTER_COMMIT` and `AFTER_ROLLBACK`).

By default, if no transaction is running the event isn't sent at all as we can't obviously honor the requested phase, but there is a fallbackExecution attribute in `@TransactionalEventListener` that tells Spring to invoke the listener immediately if there is no transaction.

NOTE	Only public methods in a managed bean can be annotated with `@EventListener` to consume events. `@TransactionalEventListener` is the annotation that provides transaction-bound event support described here.

To find out more about Spring's Event listening capabilities see the Spring reference manual and How to build Transaction aware Eventing with Spring 4.2.

8.10. 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.

8.10.1. 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.

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 3. 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
session.save(p);
```

8.10.2. 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.

Example 4. 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 5. 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 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.

8.11. Sorting and Paging

Spring Data Neo4j supports sorting and paging of results when using Spring Data's `Pageable` and `Sort` interfaces.

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);
```

NOTE The total number of pages reported by the `PagingAndSortingRepository findAll` methods are estimates and should not be relied upon for accuracy

8.12. 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 9. 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.

9.1. 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:

9.1.1. 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.

9.1.2. 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.

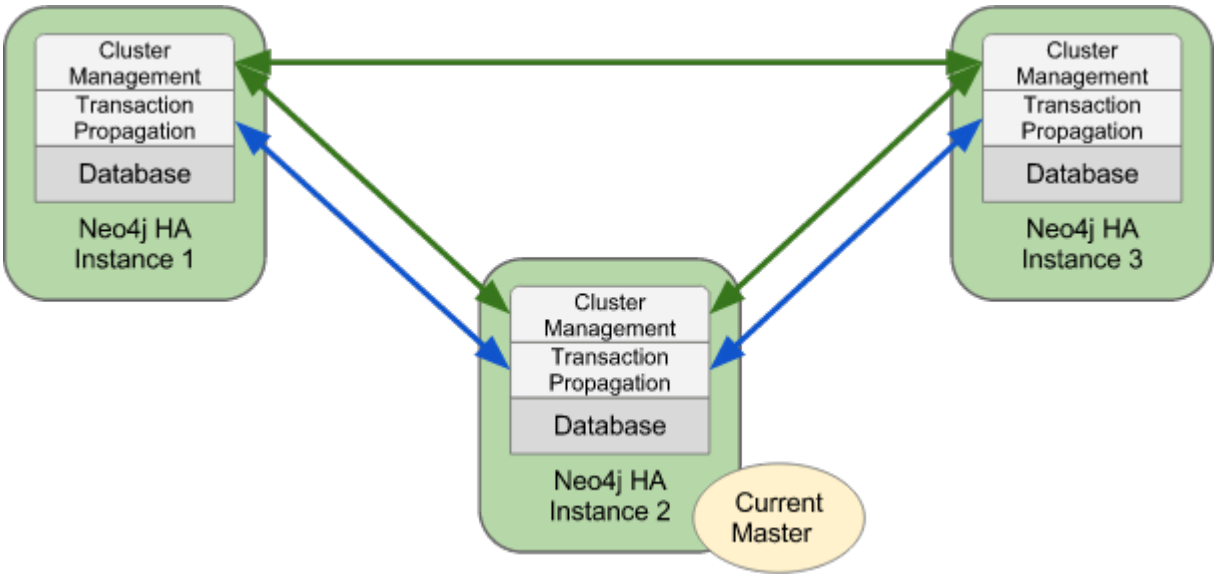
Knowing what needs to be changed means we don't need to flood Neo4j with requests to update objects that don't require updating, or create relationships that already exist. We can minimise the amount of data we send across the wire as a result, which leads to faster network interaction and fewer CPU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

Chapter 10. High Availability (HA) Environments

10.1. Configuring Spring Data Neo4j 4.2 in an HA Environment

10.1.1. 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`.

10.1.2. Read-only Transactions

As of Version 4.12.0 read-only transactions are fully supported by Spring Data Neo4j

To declare a read-only transaction specify the `readOnly` attribute in your annotation:

```
@Transactional(readOnly=true)
```

10.1.3. Drivers

The Neo4j OGM Drivers have been updated to transmit additional information about the transaction type of the current transaction to the server.

- The HttpDriver 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 READ_ONLY, but this only makes sense in an HA environment. As there is no support for HA Embedded in the OGM at the moment, the implementation is currently on hold.
- The native Bolt Driver currently has no support for READ_ONLY transactions or READ_ONLY sessions, so the implementation of this in the OGM is also on hold until this capability becomes available.

10.2. 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.

10.2.1. 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

OGM Binding via HAProxy

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

Sample haproxy.cfg

```
global
    daemon
    maxconn 256

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

frontend http-in
    bind *:80
    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
```

Migration Guide

Chapter 11. Migrating from Spring Data Neo4j 4.0 or 4.1 to Spring Data Neo4j 4.2

Spring Data Neo4j 4.2 significantly reduces complexity of configuration for application developers. There is no longer a need to extend from `Neo4jConfiguration` or define a `Session` bean. Configuration for various types of applications are described here

1. Remove any subclassing of `Neo4jConfiguration`
2. Define the `sessionFactory` bean with an instance of `SessionFactory` and the `transactionManager` bean with an instance of `Neo4jTransactionManager`. Be sure to pass the `SessionBean` into the constructor for the transaction manager.

Chapter 12. Migrating from previous versions of Spring Data Neo4j

12.1. 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.

12.2. 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`
`@MapResult`	`@QueryResult`
`@ResultColumn`	`@Property`
`Relationship Direction.BOTH`	`Relationship.UNDIRECTED`

12.3. 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

12.4. 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.

12.5. 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

12.6. Features No Longer Supported

Some features of the previous annotations have been dropped.

12.6.1. 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.

12.6.2. @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;
...
}
```

12.6.3. Cross-store Persistence

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

12.6.4. 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.

12.6.5. 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.

12.7. Changes to Neo4jTemplate

It is highly recommended for users starting new SDN projects to use `Session` directly. `Neo4jTemplate` has been kept to give upgrading users a better experience.

The `Neo4jTemplate` has been slimmed-down significantly for SDN 4. It contains the exact same methods as `Session`. In fact `Neo4jTemplate` is just a very thin wrapper with an ability to support SDN Exception Translation.

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

If you do use `Neo4jTemplate`, then you should code against its `Neo4jOperations` interface instead of the template class.

12.7.1. 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.

12.8. Indexing

The best way to retrieve start nodes for traversals and queries is by using Neo4j's integrated index facilities. Currently SDN supports Index and Constraint management but does not provide look up or merging API's. This will be provided in a future release.

12.8.1. 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.

12.8.2. 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

Appendix

Appendix A: 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`	
`NEAR`	`Near`	