Good Relationships: The Spring Data Neo4j Guide Book

Michael Hunger, Oliver Gierke

Version 3.3.1.RELEASE
2015-06-30

Table of Contents

Preface

Chapter 1. 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

Chapter 2. About this guide book

2.1. The Spring Data Neo4j Project

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

2.2. Feedback

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

If you have any feedback on Spring Data Neo4j or this book, please provide it via the SpringSource JIRA, the SpringSource NOSQL Forum, github comments or issues, or the Neo4j mailing list.

2.3. Format of the Book

This book is presented as a duplex book, a term coined by Martin Fowler. A duplex book consists of at least two parts. The first part is an easily accessible tutorial or narrative that gives the reader an overview of the topics contained in the book. It contains lots of examples and discussion topics. This part of the book is highly suited for cover-to-cover reading.

We chose a tutorial describing the creation of a web application that allows movie enthusiasts to find their favorite movies, rate them, connect with fellow movie geeks, and enjoy social features such as recommendations. The application is running on Neo4j using Spring Data Neo4j and the well-known Spring Web Stack.

The second part of the book is the classic reference documentation, 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 concrete bits of information, or to drill down into certain topics. For hackers wanting to really delve into Spring Data Neo4j, it can of course also be read cover-to-cover.

2.4. Acknowledgements

We would like to thank everyone who contributed to this book, especially Mark Pollack and Thomas Risberg, the leads of the Spring Data Project, who helped a lot during the development of the library as well as sharing great feedback about the book. Also Oliver Gierke, our local German VMWare/SpringSource engineer, who invested a lot of time discussing various aspects of the library as well as providing the superb foundations for the Spring Data Repositories. We tortured Andy Clement, the AspectJ project lead, with many questions and issues around our advanced AspectJ usage which

caused some headaches. He always quickly solved our issues and gave us excellent answers.

Many thanks to our colleagues David Montag, Andreas Kollegger and Rickard Öberg 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 project, and now also providing great forewords. 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, both in the Spring Forums as well as on the Neo4j Mailing list and on many other places on the internet for giving us feedback, reporting issues and suggesting improvements. Without that important feedback we wouldn't be where we are today. Especially Jean-Pierre Bergamin and Alfredas Chmieliauskas provided exceptional feedback and contributions.

Enjoy the book!

Tutorial

The first part of the book provides a tutorial that walks through the creation of a complete web application called cineasts.net, built with Spring Data Neo4j. Cineasts are people who love movies, and the site is a gathering place for moviegoers. For cineasts.net we decided to add a social aspect to the rating of movies, allowing friends to share their scores and get recommendations for new friends and movies.

The tutorial takes the reader through the steps necessary to create the application. It provides the configuration and code examples that are needed to understand what's happening in Spring Data Neo4j. The complete source code for the app is available on Github.

Chapter 3. Introducing our project

Allow me to introduce Cineasts.net

Once upon a time we wanted to build a social movie database. At first there was only the name: Cineasts, the movie enthusiasts who have a burning passion for movies. So we went ahead and bought the domain cineasts.net, and so we were off to a good start.

We had some ideas about the domain model too. There would obviously be actors playing roles in movies. We also needed someone to rate the movies - enter the cineast. And cineasts, being the social people they are, they wanted to make friends with other fellow cineasts. Imagine instantly finding someone to watch a movie with, or share movie preferences with. Even better, finding new friends and movies based on what you and your friends like.

When we looked for possible sources of data, IMDB was our first stop. But they're a bit expensive for our taste, charging $15k USD for data access. Fortunately, we found themoviedb.org which provides user-generated data for free. They also have liberal terms and conditions, and a nice API for retrieving the data.

We had many more ideas, but we wanted to get something out there quickly. Here is how we envisioned the final website:

Chapter 4. The Spring stack

Being Spring developers, we naturally choose components from the Spring stack to do all the heavy lifting. After all, we have the concept etched out, so we're already halfway there.

What database would fit both the complex network of cineasts, movies, actors, roles, ratings, and friends, while also being able to support the recommendation algorithms that we had in mind? We had no idea.

But hold your horses, there is this new Spring Data project, started in 2010, which brings the convenience of the Spring programming model to NOSQL databases. That should be in line with what we already know, providing us with a quick start. We had a look at the list of projects supporting the different NOSQL databases out there. Only one of them mentioned the kind of social network we were thinking of - Spring Data Neo4j for the Neo4j graph database. Neo4j's slogan of "value in relationships" plus "Enterprise NOSQL" and the accompanying docs looked like what we needed. We decided to give it a try.

4.1. Required setup

To set up the project we created a public Github account and began setting up the infrastructure for a Spring web project using Maven as the build system. So we added the dependencies for the Spring Framework libraries, added the `web.xml` for the `DispatcherServlet`, and the `applicationContext.xml` in the webapp directory.

Example 1. Project pom.xml

```
<properties>
    <spring.version>3.0.7.RELEASE</spring.version>
</properties>

<dependencies>
<dependency>
    <groupId>org.springframework</groupId>
    <!-- abbreviated for all the dependencies -->
    <artifactId>spring-(core,context,aop,aspects,tx,webmvc)</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
</dependencies>
```

Example 2. Project web.xml

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>

<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

With this setup in place we were ready for the first spike: creating a simple MovieController showing a static view. See the Spring Framework documentation for information on doing this.

Example 3. applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

 <context:annotation-config/>
 <context:spring-configured/>
 <context:component-scan base-package="org.neo4j.cineasts">
     <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
 </context:component-scan>

 <tx:annotation-driven mode="proxy"/>
</beans>
```

Example 4. dispatcherServlet-servlet.xml

```
<mvc:annotation-driven/>
<mvc:resources mapping="/images/**" location="/images/"/>
<mvc:resources mapping="/resources/**" location="/resources/"/>
<context:component-scan base-package="org.neo4j.cineasts.controller"/>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
p:prefix="/WEB-INF/views/" p:suffix=".jsp"/>
```

We spun up Tomcat in STS with the App and it worked fine. For completeness we also added Jetty to the maven-config and tested it by invoking `mvn jetty:run` to see if there were any obvious issues with the config. It all seemed to work just fine.

Chapter 5. The domain model

Setting the stage

We wanted to outline the domain model before diving into library details. We also looked at the data model of the themoviedb.org data to confirm that it matched our expectations.

In Java code this looks pretty straightforward:

Example 5. Domain model

```
class Movie {
    String id;
    String title;
    int year;
    Set<Role> cast;
}

class Actor {
    String id;
    String name;
    Set<Movie> filmography;
    Role playedIn(Movie movie, String role) { ... }
}

class Role {
    Movie movie;
    Actor actor;
    String role;
}

class User {
    String login;
    String name;
    String password;
    Set<Rating> ratings;
    Set<User> friends;
    Rating rate(Movie movie, int stars, String comment) { ... }
    void befriend(User user) { ... }
}

class Rating {
    User user;
    Movie movie;
    int stars;
    String comment;
}
```

Then we wrote some simple tests to show that the basic design of the domain is good enough so far. Just creating a movie, populating it with actors, and allowing users to rate it.

Chapter 6. Learning Neo4j

Graphs ahead

Now we needed to figure out how to store our chosen domain model in the chosen database. First we read up about graph databases, in particular our chosen one, Neo4j. The Neo4j data model consists of nodes and relationships, both of which can have key/value-style properties. What does that mean, exactly? Nodes are the graph database name for records, with property keys instead of column names. That's normal enough. Relationships are the special part. In Neo4j, relationships are first-class citizens, meaning they are more than a simple foreign-key reference to another record, relationships carry information. So we can link together nodes into semantically rich networks. This really appealed to us. Then we found that we were also able to index nodes and relationships by {key, value} pairs. We also found that we could traverse relationships both imperatively using the core API, and declaratively using a query-like Traversal Description. Besides those programmatic traversals there was the powerful graph query language called Cypher. So lots of ways of working with the graph.

We also learned that Neo4j is fully transactional and therefore upholds ACID guarantees for our data. Durability is actually a good thing and we didn't have to scale to trillions of users and movies yet. This is unusual for NOSQL databases, but easier for us to get our head around than non-transactional eventual consistency. It also made us feel safe, though it also meant that we had to manage transactions. Something to keep in mind later.

We started out by doing some prototyping with the Neo4j core API to get a feeling for how it works. And also, to see what the domain might look like when it's saved in the graph database. After adding the Maven dependency for Neo4j, we were ready to go.

Example 6. Neo4j Maven dependency

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>1.8.1</version>
</dependency>
```

Example 7. Neo4j core API (transaction code omitted)

```
enum RelationshipTypes implements RelationshipType { ACTS_IN };

GraphDatabaseService gds = new EmbeddedGraphDatabase("/path/to/store");
Node forrest=gds.createNode();
forrest.setProperty("title","Forrest Gump");
forrest.setProperty("year",1994);
gds.index().forNodes("movies").add(forrest,"id",1);

Node tom=gds.createNode();
tom.setProperty("name","Tom Hanks");

Relationship role=tom.createRelationshipTo(forrest,ACTS_IN);
role.setProperty("role","Forrest");

Node movie=gds.index().forNodes("movies").get("id",1).getSingle();
assertEquals("Forrest Gump", movie.getProperty("title"));
for (Relationship role : movie.getRelationships(ACTS_IN,INCOMING)) {
    Node actor=role.getOtherNode(movie);
    assertEquals("Tom Hanks", actor.getProperty("name"));
    assertEquals("Forrest", role.getProperty("role"));
}
```

Chapter 7. Spring Data Neo4j

Conjuring magic

So far it had all been pure Spring Framework and Neo4j. However, using the Neo4j code in our domain classes polluted them with graph database details. For this application, we wanted to keep the domain classes clean. Spring Data Neo4j promised to do the heavy lifting for us, so we continued investigating it.

Spring Data Neo4j comes with two mapping modes. The more powerful one depends heavily on AspectJ, see AspectJ details, so we ignored it for the time being. The simple direct POJO-mapping copies the data out of the graph and into our entities. Good enough for a web-application like ours.

The first step was to configure Maven:

Example 8. Spring Data Neo4j Maven configuration

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j</artifactId>
  <version>2.1.0.RELEASE</version>
</dependency>
```

The Spring context configuration was even easier, thanks to a provided namespace:

Example 9. Spring Data Neo4j context configuration

```
<beans xmlns="http://www.springframework.org/schema/beans" ...
       xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
       xsi:schemaLocation="... http://www.springframework.org/schema/data/neo4j
       http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">
    ...
    <neo4j:config storeDirectory="data/graph.db"/>
    ...
</beans>
```

Chapter 8. Annotating the domain

Decorations

Looking at the Spring Data Neo4j documentation, we found a simple Hello World example and tried to understand it. We also spotted a compact reference card which helped us a lot. The entity classes were annotated with `@NodeEntity`. That was simple, so we added the annotation to our domain classes too. Entity classes representing relationships were instead annotated with `@RelationshipEntity`. Property fields were taken care of automatically. The only additional field we had to provide for all entities was an id-field to store the node- and relationship-ids.

Example 10. Movie class with annotation

```
@NodeEntity
class Movie {
    @GraphId Long nodeId;
    String id;
    String title;
    int year;
    Set<Role> cast;
}
```

It was time to put our entities to the test. How could we now be assured that an attribute really was persisted to the graph store? We wanted to load the entity and check the attribute. Either we could have a `Neo4jTemplate` injected and use its `findOne(id,type)` method to load the entity. Or use a more versatile `Repository`. The same goes for persisting entities, both `Neo4jTemplate` or the `Repository` could be used. We decided to keep things simple for now.

So here's what our test ended up looking like:

Example 11. First test case

```
@Autowired Neo4jTemplate template;

@Test @Transactional public void persistedMovieShouldBeRetrievableFromGraphDb() {
    Movie forrestGump = template.save(new Movie("Forrest Gump", 1994));
    Movie retrievedMovie = template.findOne(forrestGump.getNodeId(), Movie.class);
    assertEquals("retrieved movie matches persisted one", forrestGump,
retrievedMovie);
    assertEquals("retrieved movie title matches", "Forrest Gump", retrievedMovie
.getTitle());
}
```

As Neo4j is transactional, we have to provide the transactional boundaries for mutating operations.

Chapter 9. Indexing

Do I know you?

There is an @Indexed annotation for fields. We wanted to try this out, and use it to guide the next test. We added @Indexed to the `id` field of the Movie class. This field is intended to represent the external ID that will be used in URIs and will be stable across database imports and updates. That's why we also declare it as unique. This time we went with a simple GraphRepository to retrieve the indexed movie.

Example 12. Exact Indexing for Movie id

```
@NodeEntity class Movie {
    @Indexed(unique=true) String id;
    String title;
    int year;
}

@Autowired Neo4jTemplate template;

@Test @Transactional
  public void persistedMovieShouldBeRetrievableFromGraphDb() {
    int id = 1;
    Movie forrestGump = template.save(new Movie(id, "Forrest Gump", 1994));
    GraphRepository<Movie> movieRepository =
                                template.repositoryFor(Movie.class);
    Movie retrievedMovie = movieRepository.findByPropertyValue("id", id);
    assertEqual("retrieved movie matches persisted one", forrestGump, retrievedMovie
);
    assertEqual("retrieved movie title matches", "Forrest Gump", retrievedMovie
.getTitle());
  }
```

Chapter 10. Repositories

Serving a good cause

We wanted to add repositories with domain-specific operations. Interestingly there was support for a very advanced repository infrastructure. You just declare an entity specific repository interface and get all commonly used methods for free without implementing any of boilerplate code.

So we started by creating a movie-related repository, simply by creating an empty interface.

Example 13. Movie repository

```
package org.neo4j.cineasts.repository;
public interface MovieRepository extends GraphRepository<Movie> {}
```

Then we enabled repository support in the Spring context configuration by simply adding:

Example 14. Repository context configuration

```
<neo4j:repositories base-package="org.neo4j.cineasts.repository"/>
```

Besides the existing repository operations (like CRUD, and many standard queries) it was possible to declare custom methods, which we explored later. Those methods' names could be more domain centric and expressive than the generic operations. For simple use-cases like finding by id's this is good enough. So we first let Spring autowire our `MovieController` with the `MovieRepository`. That way we could perform simple persistence operations.

Example 15. Usage of a repository

```
@Autowired MovieRepository repo;
...
  Movie movie = repo.findByPropertyValue("id",movieId);
```

We went on exploring the repository infrastructure. A very cool feature was something that we so far only heard about from Grails developers. Deriving queries from method names. Impressive! So we had a more explicit method for the id lookup.

Example 16. Derived movie-repository query method

```
public interface MovieRepository extends GraphRepository<Movie> {
  Movie getMovieById(String id);
}
```

In our wildest dreams we imagined the method names we would come up with, and what kinds of queries those could generate. But some, more complex queries would be cumbersome to read and write. So in those cases it is better to just annotate the finder method. We did this much later, and just wanted to give you a peek into the future. There is much more, you can do with repositories, it is worthwhile to explore.

Example 17. Annotated movie-repository query method

```
public interface MovieRepository extends GraphRepository<Movie> {
 @Query("start user=node:User({0}) match user-[r:RATED]->movie return movie order by
r.stars desc limit 10")
 Iterable<Movie> getTopRatedMovies(User uer);
}
```

Chapter 11. Relationships

A convincing act

Our application was not very much fun yet, just storing movies and actors. After all, the power is in the relationships between them. Fortunately, Neo4j treats relationships as first class citizens, allowing them to be addressed individually and have properties assigned to them. That allows for representing them as entities if needed.

11.1. Creating relationships

Relationships without properties ("anonymous" relationships) don't require any `@RelationshipEntity` classes. "Unfortunately" we had none of those, because our relationships were richer. Therefore we went with the `Role` relationship between `Movie` and `Actor`. It had to be annotated with `@RelationshipEntity` and the `@StartNode` and `@EndNode` had to be marked. So our Role looked like this:

Example 18. Role class

```
@RelationshipEntity
class Role {
    @StartNode Actor actor;
    @EndNode Movie movie;
    String role;
}
```

When writing a test for the `Role` we tried to create the relationship entity just by instantiating it with `new` and saving it with the template, but we got an exception saying that it misses the relationship-type.

We had to add it to the `@RelationshipEntity` as an attribute (or as a @RelationshipType annotated field in the RelationshipEntity). Another way to create instances of relationship-entities is to use the methods provided by the template, like `createRelationshipBetween`.

Example 19. Relating actors to movies

```
@RelationshipEntity(type="ACTS_IN")
class Role {
    @StartNode Actor actor;
    @EndNode Movie movie;
    String role;
}
class Actor {
...
    public Role playedIn(Movie movie, String roleName) {
        Role role = new Role(this, movie, roleName);
        this.roles.add(role);
        return role;
    }
}

Role role = tomHanks.playedIn(forrestGump, "Forrest Gump");

// either save the actor
template.save(tomHanks);
// or the role
template.save(role);

// alternative approach
Role role = template.createRelationshipBetween(actor,movie,
                        Role.class, "ACTS_IN");
```

Saving just the actor would take care of relationships with the same type between two entities and remove the duplicates. Whereas just saving the role happily creates another relationship with the same type.

11.2. Accessing related entities

Now we wanted to find connected entities. We already had fields for the relationships in both classes. It was time to annotate them correctly. The Neo4j relationship type and direction were easy to figure out. The direction even defaulted to outgoing, so we only had to specify it for the movie. If we want to use the same relationship between the two entities we have to make sure to provide a dedicated type, otherwise the field-names would be used resulting in different relationships.

Example 20. @RelatedTo usage

```
@NodeEntity
class Movie {
    @Indexed(unique=true) String id;
    String title;
    int year;
    @RelatedTo(type = "ACTS_IN", direction = Direction.INCOMING)
    Set<Actor> cast;
}

@NodeEntity
class Actor {
    @Indexed(unique=true) int id;
    String name;
    @RelatedTo(type = "ACTS_IN")
    Set<Movie> movies;

    public Role playedIn(Movie movie, String roleName) {
        return new Role(this,movie, roleName);
    }
}
```

Changes to the collections of related entities are reflected into the graph on saving of the entity.

We made sure to add some tests for using the relationshhips, so we were assured that the collections worked as advertised.

11.3. Accessing the relationship entities

But we still couldn't access the Role relationship entities themselves. It turned out that there was a separate annotation `@RelatedToVia` for accessing the actual relationship entities. And we could declare the field as an `Iterable<Role>`, with read-only semantics or on a `Collection` or `Set<Role>` field with modifying semantics. So off we went, creating our first real relationship (just kidding).

To have the collections of relationships being read eagerly during the loading of the Movie we have to annotate it with the @Fetch annotation. Otherwise Spring Data Neo4j refrains from following relationships automatically. The risk of loading the whole graph into memory would be too high.

Example 21. @RelatedToVia usage

```
@NodeEntity
class Movie {
    @Indexed(unique=true) String id;
    String title;
    int year;

    @Fetch @RelatedToVia(type = "ACTS_IN", direction = Direction.INCOMING)
    Iterable<Roles> roles;
}
```

After watching the tests pass, we were confident that the changes to the relationship fields were really stored to the underlying relationships in the graph. We were pretty satisfied with persisting our domain.

Chapter 12. Get it running

Curtains up!

Now we had a pretty complete application. It was time to put it to the test.

12.1. Populating the database

Before we opened the gates we needed to add some movie data. So we wrote a small class for populating the database which could be called from our controller. A simple `/populate` endpoint for the controller that called it would be enough for now.

Example 22. Populating the database - Controller

```
@Service
public class DatabasePopulator {

    @Transactional
    public List<Movie> populateDatabase() {
        Actor tomHanks = new Actor("1", "Tom Hanks");
        Movie forrestGump = new Movie("1", "Forrest Gump");
        tomHanks.playedIn(forrestGump,"Forrest");
        template.save(forrestGump);
        return asList(forrestGump);
    }
}

@Controller
public class MovieController {

    @Autowired private DatabasePopulator populator;

    @RequestMapping(value = "/populate", method = RequestMethod.POST)
    public String populateDatabase(Model model) {
        Collection<Movie> movies = populator.populateDatabase();
        model.addAttribute("movies",movies);
        return "/movies/list";
    }
}
```

Accessing the URI we could see the list of movies we had added.

12.2. Inspecting the datastore

Being the geeks we are, we also wanted to inspect the raw data in the database. Reading the Neo4j docs, there were a couple of different ways of going about this.

12.2.1. Neoclipse visualization

First we tried Neoclipse, an Eclipse RCP application that opens an existing graph store and visualizes its content. After getting an exception about concurrent access, we learned that we have to use Neoclipse in read-only mode when our webapp was still running. Good to know.

12.2.2. The Neo4j Shell

For console junkies there was also a shell that was able to connect to a running Neo4j instance (if it was started with the `enable_remote_shell=true` parameter), or reads an existing graph store directly.

Example 23. Starting the Neo4j Shell

```
bash# neo4j-shell -readonly -path data/graph.db
bash# neo4j-shell -readonly -port 1337
```

The shell was very similar to a standard Bash shell. We were able to `cd` to between the nodes, and `ls` the relationships and properties. There were also more advanced commands for indexing, queries and traversals.

Example 24. Neo4j Shell usage

```
neo4j-sh[readonly] (0)$ help
Available commands: index dbinfo ls rm alias set eval mv gsh env rmrel mkrel
                    trav help pwd paths ... man cd
Use man <command> for info about each command.

neo4j-sh[readonly] (0)$ index --cd -g User login micha

neo4j-sh[readonly] (Micha,1)$ ls
*__type__ =[org.neo4j.cineasts.domain.User]
*login    =[micha]
*name     =[Micha]
*roles    =[ROLE_ADMIN,ROLE_USER]
(me) --[FRIEND]-> (Olliver,2)
(me) --[RATED]-> (The Matrix,3)

neo4j-sh[readonly] (Micha,1)$ ls 2
*__type__ =[org.neo4j.cineasts.domain.User]
*login    =[ollie]
*name     =[Olliver]
*roles    =[ROLE_USER]
(Olliver,2) <-[FRIEND]-- (me)

neo4j-sh[readonly] (Micha,1)$ cd 3

neo4j-sh[readonly] (The Matrix,3)$ ls
*__type__     =[org.neo4j.cineasts.domain.Movie]
*description  =[Neo is a young software engineer and part-time hacker who is singled
...]
*genre        =[Action]
*homepage     =[http://whatisthematrix.warnerbros.com/]
...
*studio       =[Warner Bros. Pictures]
*tagline      =[Welcome to the Real World.]
*title        =[The Matrix]
*trailer      =[http://www.youtube.com/watch?v=UM5yepZ21pI]
*version      =[324]
(me) <-[ACTS_IN]-- (Marc Aden,19)
(me) <-[ACTS_IN]-- (David Aston,18)
...
(me) <-[ACTS_IN]-- (Keanu Reeves,6)
(me) <-[DIRECTED]-- (Andy Wachowski,5)
(me) <-[DIRECTED]-- (Lana Wachowski,4)
(me) <-[RATED]-- (Micha,1)
```

Chapter 13. Web views

Showing off

After having put some data in the graph database, we also wanted to show it to the user. Adding the controller method to show a single movie with its attributes and cast in a JSP was straightforward. It basically just involved using the repository to look the movie up and add it to the model, and then forwarding to the `/movies/show` view and voilá.

Example 25. Controller for showing movies

```
@RequestMapping(value = "/movies/{movieId}",
method = RequestMethod.GET, headers = "Accept=text/html")
public String singleMovieView(final Model model, @PathVariable String movieId) {
    Movie movie = repository.findById(movieId);
    model.addAttribute("id", movieId);
    if (movie != null) {
        model.addAttribute("movie", movie);
        model.addAttribute("stars", movie.getStars());
    }
    return "/movies/show";
}
```

Example 26. Populating the database - JSP /movies/show

```
<%@ page session="false" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:choose>
<c:when test="${not empty movie}">
<h2>${movie.title} (${stars} Stars)</h2>
<c:if test="${not empty movie.roles}">
<ul>
<c:forEach items="${movie.roles}" var="role">
<li>
<a href="/actors/${role.actor.id}"><c:out value="${role.actor.name}" /> as
<c:out value="${role.name}" /></a><br/>
</li>
</c:forEach>
</ul>
</c:if>
</c:when>
<c:otherwise>
No Movie with id ${id} found!
</c:otherwise>
</c:choose>
```

The UI had now evolved to this:

13.1. Searching

The next thing was to allow users to search for movies, so we needed some fulltext search capabilities. As the default index provider implementation of Neo4j is based on Apache Lucene, we were delighted to see that fulltext indexes were supported out of the box.

We happily annotated the title field of the Movie class with `@Indexed(type = FULLTEXT)`. Next thing we got an exception telling us that we had to specify a separate index name. So we simply changed it to `@Indexed(type = FULLTEXT, indexName = "search")`.

With derived finder methods, finding things became easy. By simply declaring a finder-method name that expressed the required properties, it worked without annotations. Cool stuff and you could even tell it that it should return pages of movies, its size and offset specified by a `Pageable` which also contains sort information. Using the `like` operator indicates that fulltext search should be used, instead of an exact search.

Example 27. Searching for movies

```
public interface MovieRepository ... {
    Movie findById(String id);
    Page<Movie> findByTitleLike(String title, Pageable page);
    Slice<Movie> findAll(Pageable page);
}
```

13.2. Listing results

We then used this result in the controller to render a page of movies, driven by a search box. The movie properties and the cast were accessible through the getters in the domain classes.

Example 28. Search controller

```
@RequestMapping(value = "/movies",
method = RequestMethod.GET, headers = "Accept=text/html")
public String findMovies(Model model, @RequestParam("q") String query) {
    Page<Movie> movies = repository.findByTitleLike(query, new PageRequest(0,20));
    model.addAttribute("movies", movies);
    model.addAttribute("query", query);
    return "/movies/list";
}
```

Example 29. Search Results JSP

```
<h2>Movies</h2>

<c:choose>
    <c:when test="${not empty movies}">
        <dl class="listings">
        <c:forEach items="${movies}" var="movie">
            <dt>
                <a href="/movies/${movie.id}"><c:out value="${movie.title}" /></a>
<br/>
            </dt>
            <dd>
                <c:out value="${movie.description}" escapeXml="true" />
            </dd>
        </c:forEach>
        </dl>
    </c:when>
    <c:otherwise>
        No movies found for query &amp;quot;${query}&amp;quot;.
    </c:otherwise>
</c:choose>
```

The UI now looked like this:

Chapter 14. Adding social

Movies 2.0

So far, the website had only been a plain old movie database. We now wanted to add a touch of social to it.

14.1. Users

So we started out by taking the User class that we'd already coded and made it a full-fledged Spring Data Neo4j entity. We added the ability to create friends and to rate movies. With that we also added a simple UserRepository that was able to look up users by ID.

The relationships of the user are his friends and the movie-ratings which is implemented with a `Rating` Relationship-Entity. This time we used a different approach (for educational and curiosity purposes) to create the `Rating` relationships. The `createRelationshipBetween` operation of the Neo4jTemplate was our matchmaker of choice.

Example 30. Social entities

```
@NodeEntity
class User {
    @Indexed(unique=true) String login;
    String name;
    String password;

    @RelatedToVia(type = RATED)
    @Fetch Set<Rating> ratings;

    @RelatedTo(type = "FRIEND", direction=Direction.BOTH)
    @Fetch Set<User> friends;

    public Rating rate(Neo4jOperations template, Movie movie, int stars, String
comment) {
        final Rating rating = template.createRelationshipBetween(this, movie, Rating
.class, RATED, false);
        rating.rate(stars, comment);
        return template.save(rating);
    }

    public void addFriend(User user) {
        this.friends.add(user);
    }
}

@RelationshipEntity
class Rating {
    @StartNode User user;
    @EndNode Movie movie;
    int stars;
    String comment;
    public Rating rate(int stars, String comment) {
       this.stars = stars; this.comment = comment;
       return this;
    }
}
```

We extended the DatabasePopulator to add some users and ratings to the initial setup.

Example 31. Populate users and ratings

```
@Transactional
public List<Movie> populateDatabase() {
    Actor tomHanks = new Actor("1", "Tom Hanks");
    Movie forestGump = new Movie("1", "Forrest Gump");
    tomHanks.playedIn(forestGump, "Forrest");
    template.save(tomHanks);

    User me = template.save(new User("micha", "Micha", "password"));
    Rating awesome = me.rate(template, forestGump, 5, "Awesome");

    User ollie = template.save(new User("ollie", "Oliver", "password"));
    ollie.rate(template,forestGump, 2, "ok");
    me.addFriend(ollie);
    template.save(me);
    return asList(forestGump);
}
```

14.2. Ratings for movies

We also put a ratings field into the Movie class to be able to get a movie's ratings, and also a method to average its star rating.

Example 32. Getting the rating of a movie

```
class Movie {
    ...

    @RelatedToVia(type="RATED", direction = Direction.INCOMING)
    @Fetch Iterable<Rating> ratings;

    public int getStars() {
        int stars = 0, count = 0;
        for (Rating rating : ratings) {
            stars += rating.getStars(); count++;
        }
        return count == 0 ? 0 : stars / count;
    }
}
```

Fortunately our tests highlighted the division by zero error when calculating the stars for a movie

without ratings. The next steps were to add this information to the movie presentation in the UI, and creating a user profile page. But for that to happen, users must first be able to log in.

Chapter 15. Adding Security

Protecting assets

To handle an active user in the webapp we had to put it in the session and add login and registration pages. Of course the pages that were only meant for logged-in users had to be secured as well.

Being Spring users, we naturally used Spring Security for this. We wrote a simple `UserDetailsService` by extending a repository with a custom implementation that takes care of looking up the users and validating their credentials. The config is located in a separate `applicationContext-security.xml`. But first, as always, Maven and `web.xml` setup.

Example 33. Spring Security pom.xml

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${spring.version}</version>
</dependency>
```

Example 34. Spring Security web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/applicationContext-security.xml
        /WEB-INF/applicationContext.xml
    </param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>

<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Example 35. Spring Security applicationContext-security.xml

```
<security:global-method-security secured-annotations="enabled">
</security:global-method-security>

<security:http auto-config="true" access-denied-page="/auth/denied">
    <security:intercept-url pattern="/admin/*" access="ROLE_ADMIN"/>
    <security:intercept-url pattern="/import/*" access="ROLE_ADMIN"/>
    <security:intercept-url pattern="/user/*" access="ROLE_USER"/>
    <security:intercept-url pattern="/auth/login" access=
"IS_AUTHENTICATED_ANONYMOUSLY"/>
    <security:intercept-url pattern="/auth/register" access=
"IS_AUTHENTICATED_ANONYMOUSLY"/>
    <security:intercept-url pattern="/**" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
    <security:form-login login-page="/auth/login"
authentication-failure-url="/auth/login?login_error=true"
    default-target-url="/user"/>
    <security:logout logout-url="/auth/logout" logout-success-url="/" invalidate-
session="true"/>
</security:http>

<security:authentication-manager>
    <security:authentication-provider user-service-ref="userRepository">
        <security:password-encoder hash="md5">
            <security:salt-source system-wide="cewuiqwzie"/>
        </security:password-encoder>
    </security:authentication-provider>
</security:authentication-manager>
```

Example 36. CinceastUserDetailsService interface and UserRepository custom implementation

```
public interface CineastsUserDetailsService extends UserDetailsService {
    @Override
    CineastsUserDetails loadUserByUsername(String login)
                throws UsernameNotFoundException, DataAccessException;

    User getUserFromSession();

    @Transactional
    Rating rate(Movie movie, User user, int stars, String comment);

    @Transactional
    User register(String login, String name, String password);

    @Transactional
    void addFriend(String login, final User userFromSession);
}

public interface UserRepository extends GraphRepository<User>,
        RelationshipOperationsRepository<User>,
        CineastsUserDetailsService {

    User findByLogin(String login);
}

public class UserRepositoryImpl implements CineastsUserDetailsService {

  @Autowired private Neo4jOperations template;

  @Override
  public CineastsUserDetails loadUserByUsername(String login)
         throws UsernameNotFoundException, DataAccessException {
      final User user = findByLogin(login);
      if (user==null) throw
         new UsernameNotFoundException("Username not found: "+login);
      return new CineastsUserDetails(user);
  }

  private User findByLogin(String login) {
      return template.lookup(User.class,"login",login)
                     .to(User.class).single();
  }

  @Override
  public User getUserFromSession() {
      SecurityContext context = SecurityContextHolder.getContext();
      Authentication authentication = context.getAuthentication();
      Object principal = authentication.getPrincipal();
```

```
      if (principal instanceof CineastsUserDetails) {
          CineastsUserDetails userDetails = (CineastsUserDetails) principal;
          return userDetails.getUser();
      }
      return null;
  }
}

public class CineastsUserDetails implements UserDetails {
    private final User user;

    public CineastsUserDetails(User user) {
        this.user = user;
    }

    @Override
    public Collection<GrantedAuthority> getAuthorities() {
        User.Roles[] roles = user.getRoles();
        if (roles ==null) return Collections.emptyList();
        return Arrays.<GrantedAuthority>asList(roles);
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getLogin();
    }

    ...
    public User getUser() {
        return user;
    }
}
```

Any logged-in user was now available in the session, and could be used for all the social interactions. The remaining work for this was mainly adding controller methods and JSPs for the views. We used the helper method `getUserFromSession()` in the controllers to access the logged-in user and put it in the model for rendering. Here's what the UI had evolved to:

Chapter 16. More UI

Oh the glamour

To create a nice user experience, we wanted to have a nice looking app. Not something that looked like a toddler made it. So we got some user experience people involved and the results were impressive. This sections presents some of the remaining screen shots of Cineasts.net.

Chapter 17. Importing Data

The dusty archives

It was now time to pull the data from themoviedb.org. Registering there and getting an API key was simple, as was using the API on the command-line with `curl`. Looking at the JSON returned for movies and people, we decided to enhance our domain model and add some more fields to enrich the UI.

Example 37. JSON movie response

```
[{"popularity":3,
"translated":true, "adult":false, "language":"en",
"original_name":"[Rec]", "name":"[Rec]", "alternative_name":"[REC]",
"movie_type":"movie",
"id":8329, "imdb_id":"tt1038988", "url":"http://www.themoviedb.org/movie/8329",
"votes":11, "rating":7.2,
"status":"Released",
"tagline":"One Witness. One Camera",
"certification":"R",
"overview":"\"REC\" turns on a young TV reporter and her cameraman who cover the
night shift
 at the local fire station...
"keywords":["terror", "lebende leichen", "obsession", "camcorder", "firemen",
"reality tv ",
 "bite", "cinematographer",
"attempt to escape", "virus", "lodger", "live-reportage", "schwerverletzt"],
"released":"2007-08-29",
"runtime":78,
"budget":0,
"revenue":0,
"homepage":"http://www.3l-filmverleih.de/rec",
"trailer":"http://www.youtube.com/watch?v=YQUkX_XowqI",
"genres":[{"type":"genre",
"url":"http://themoviedb.org/genre/horror",
"name":"Horror",
"id":27}],
"studios":[{"url":"http://www.themoviedb.org/company/2270", "name":"Filmax Group",
"id":2270}],
"languages_spoken":[{"code":"es", "name":"Spanish", "native_name":"Espa\u00f1ol"}],
"countries":[{"code":"ES", "name":"Spain", "url":
"http://www.themoviedb.org/country/es"}],
"posters":[{"image":{"type":"poster",
"size":"original", "height":1000, "width":706,
"url":"http://cf1.imgobject.com/posters/3a0/4cc8df415e73d650240003a0/rec-
original.jpg",
"id":"4cc8df415e73d650240003a0"}},
....
"cast":[{"name":"Manuela Velasco",
"job":"Actor", "department":"Actors",
"character":"Angela Vidal",
"id":34793, "order":0, "cast_id":1,
"url":"http://www.themoviedb.org/person/34793",
"profile":"http://cf1.imgobject.com/profiles/390/.../manuela-velasco-thumb.jpg"},
...
{"name":"Gl\u00f2ria Viguer",
"job":"Costume Design", "department":"Costume \u0026 Make-Up",
"character":"",
```

```
"id":54531, "order":0, "cast_id":21,
"url":"http://www.themoviedb.org/person/54531",
"profile":""}],
"version":150, "last_modified_at":"2011-02-20 23:16:57"}]
```

Example 38. JSON actor response

```
[{"popularity":3,
"name":"Glenn Strange", "known_as":[{"name":"George Glenn Strange"}, {"name":"Glen
Strange"},
{"name":"Glen 'Peewee' Strange"}, {"name":"Peewee Strange"}, {"name":"'Peewee'
Strange"}],
"id":30112,
"biography":"",
"known_movies":4,
"birthday":"1899-08-16", "birthplace":"Weed, New Mexico, USA",
"url":"http://www.themoviedb.org/person/30112",
"filmography":[{"name":"Bud Abbott Lou Costello Meet Frankenstein",
"id":3073,
"job":"Actor", "department":"Actors",
"character":"The Frankenstein Monster",
"cast_id":23,
"url":"http://www.themoviedb.org/movie/3073",
"poster":"http://cf1.imgobject.com/posters/4ca/.../bud-abbott-lou-costello-meet-
frankenstein-cover.jpg",
"adult":false, "release":"1948-06-15"},
...],
"profile":[],
"version":19, "last_modified_at":"2011-03-07 13:02:35"}]
```

For the import process we created a separate importer using Jackson (a JSON library) to fetch and parse the data, and then some transactional methods in the `MovieDbImportService` to actually import it as movies, roles, and actors. The importer used a simple caching mechanism to keep downloaded actor and movie data on the filesystem, so that we didn't have to overload the remote API. In the code below you can see that we've changed the actor to a person so that we can also accommodate the other folks that participate in movie production.

Example 39. Importing the data

```
@Transactional
public Movie importMovie(String movieId) {
  Movie movie = movieRepository.findById(movieId);
  if (movie == null) { // Not found: Create fresh
      movie = new Movie(movieId,null);
  }

  Map data = loadMovieData(movieId);
  if (data.containsKey("not_found")) throw
     new RuntimeException("Data for Movie "+movieId+" not found.");
  movieDbJsonMapper.mapToMovie(data, movie);
  movieRepository.save(movie);
  relatePersonsToMovie(movie, data);
  return movie;
}

private void relatePersonsToMovie(Movie movie, Map data) {
  Collection<Map> cast = (Collection<Map>) data.get("cast");
  for (Map entry : cast) {
    String id = "" + entry.get("id");
    String jobName = (String) entry.get("job");
    Roles job = movieDbJsonMapper.mapToRole(jobName);
    if (job==null) {
      continue;
    }
    switch (job) {
      case DIRECTED:
        final Director director = doImportPerson(id, new Director(id));
        director.directed(movie);
        directorRepository.save(director);
        break;
      case ACTS_IN:
        final Actor actor = doImportPerson(id, new Actor(id));
        actor.playedIn(movie, (String) entry.get("character"));
        actorRepository.save(actor);
        break;
    }
  }
}

public void mapToMovie(Map data, Movie movie) {
  movie.setTitle((String) data.get("name"));
  movie.setLanguage((String) data.get("language"));
  movie.setTagline((String) data.get("tagline"));
  movie.setReleaseDate(toDate(data, "released", "yyyy-MM-dd"));
...
  movie.setImageUrl(selectImageUrl((List<Map>) data.get("posters"), "poster", "mid")
```

```
);
}
```

The last part involved adding a protected URI to the MovieController to allow importing ranges of movies. During testing, it became obvious that the calls to themoviedb.org were a limiting factor. As soon as the data was stored locally, the Neo4j import was a sub-second deal.

Chapter 18. Recommendations

Movies! Friends! Bargains!

In the last part of this exercise we wanted to add recommendations to the app. One obvious recommendation was movies that our fiends liked.

There was this query language called Cypher that looked a bit like SQL but expressed graph matching queries. So we gave it a try, using the `neo4j-shell`, to incrementally expand the query, just by declaring what relationships we wanted to be taken into account and which properties of nodes and relationships to filter and sort on.

Example 40. Cypher based movie recommendation on Repository

```
interface MovieRepository extends GraphRepository<Movie> {
@Query("
match (user)-[:FRIEND]-(friend)-[r:RATED]->(movie)
where id(user) = {0}
return movie
order by avg(r.stars) desc, count(*) desc
limit 10
")
    Iterable<Movie> recommendMovies(User me);
}
```

But we didn't have enough friends, so it was time to get some suggested. That would be like-minded cineasts that rated movies similarly to us. Again Cypher to the rescue, this time only a bit more complex. Something that became obvious with both queries is that graph queries are always local, so they start from a node, or set of nodes or relationships, and then expand outwards from there.

Example 41. Cypher - Friend Recommendation on Repository

```
interface UserRepository extends GraphRepository<User> {
@Query("
start user=node({0})
match (user)-[r:RATED]->(movie)<-[r2:RATED]-(likeminded),
where id(user) = {0} and r.stars > 3 and r2.stars >= 3
and not (user)-[:FRIEND]-(likeminded)
with likeminded, count(*)
order by count(*) desc
limit 10
return likeminded
")
    Iterable<User> suggestFriends(User me);
}
```

The controllers simply called these methods, added their results to the model, and the view rendered the recommendations alongside the user's own ratings.

Chapter 19. Neo4j Server

Remotely related

Right now our application was running with the embedded mode of Neo4j which was fine and highly performant. In certain environments you don't have the luxury of file-system access for your webapps and have to talk to a remote database service instead. Neo4j can also run as a server. It exposes its operations via a HTTP based REST API.

We decided to have a look, to be at least knowledgeable about this deployment scenario. We were aware of the difference of local, in-memory calls and higher latency network hops. That would be something we would also take into careful consideration.

19.1. Getting Neo4j-Server

Getting the Neo4j-Server was easy, we just went to neo4j.org and downloaded the latest version. Starting it on the command-line (or installing it as a service) was a no-brainer as well.

We copied our store-directory into the `data/graph.db` directory of the server and started it up again. The admin console of Neo4j-Server, called 'web-admin' is pretty. Using JavaScript, it renders the graph visually in a highly configurable way. It also gave us the possibility to issue queries over a console, another handy feature.

So, how would we get our app connected to this server? It turned out the changes in configuration and setup where minimal. Spring Data Neo4j already came with a module that took care of the remote protocol. We added that maven dependency and changed the graph database used in the Spring Configuration.

Example 42. Maven Dependency

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j-rest</artifactId>
    <version>2.1.0.RELEASE</version>
</dependency>
```

Example 43. Spring Config

```
<neo4j:config graphDatabaseService="graphDatabaseService"/>
<bean id="graphDatabaseService"
class="org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
  <constructor-arg index="0" value="http://localhost:7474/db/data" />
</bean>
```

After those two changes we restarted the app, and ... it worked. The transparent handling of the remote API was impressive. We learned that it uses a library called java-rest-binding under the hood which is also usable without the Spring Framework.

Of course we noticed performance implications. Especially after moving the server to a remote machine. It turned out that the server supported remote execution of many operations, allowing us to run the graph traversal and querying inside the server. That means looking at our graph interactions and changing them in a way that switched from the transparent, direct graph access via the entities to a different interaction pattern.

We looked into the different modes of remotely executed operations and found traversals, Cypher queries and index lookups. Most of them already matched our needs but the Cypher approaches were best suited, because they also handled index operations and allowed to return partial attribute sets and subgraphs.

So we looked at our use-case (aka page)-based interactions with the graph entities and converted them to Cypher queries on repositories where appropriate, measuring the performance improvements as we went.

There was also a nice mechanism of mapping Cypher query results to Domain Concepts. You just had to declare and annotate an interface that represents the query results as domain entities and the nodes and relationships returned by Cypher were converted into the appropriate entities.

Example 44. Example of query result mapping

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

    @Query("START movie=node:Movie(id={0})
            MATCH movie-[rating?:rating]->(),
                  movie<-[:ACTS_IN]-actor
            RETURN movie, COLLECT(actor), AVG(rating.stars)")
    MovieData getMovieData(String movieId);

    @MapResult
    public interface MovieData {
        @ResultColumn("movie")
        Movie getMovie();

        @ResultColumn("AVG(rating.stars)")
        Double getRating();

        @ResultColumn("COLLECT(actor)")
        Iterable<Actor> getCast();
    }
}
```

This allowed us to get all the data needed for rendering a page in a single call to the server, greatly diminishing the chatter between the client and the server.

19.2. Other approaches

Another approach to using the Neo4j-Server would be to write a custom server extension using the `SpringPluginInitializer` provided by `spring-data-neo4j-rest`. This extension would use the well known entities and approaches as it runs inside the server atop an embedded graph database. From the extension we would expose custom, domain and use-case oriented REST endpoints that could then be consumed by any kind of webapp, even a pure Javascript based browser app.

Chapter 20. Conclusion

To new frontiers

Pretty neat. We were satisfied with what we got here, with little effort and high performance. Lots of opportunities to expand the social movie database showed up during development. Like adding more social features like tagging, communication streams, location based features (cinemas) and much more.

But we leave you with that as an exercise to enjoy and explore. Thanks for following the tutorial and make sure to get back to us with suggestions for improvements or reports about unexpected behaviours at the discussion forums, or the issue tracker.

Reference Documentation

This part of the Spring Data Neo4j Guide book provides the reference documentation. It details many aspects of the tutorial and also explains concepts that were only just mentioned there.

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

Whenever you look for the means to employ the full power of the Spring Data Neo4j library you find your answers in the reference section. If you don’t, please inform us about missing or incorrect content so that we can fix that.

Reference Documentation

Spring Data and Spring Data Neo4j

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, Riak), document (MongoDB) and relational (Oracle) databases. Mark Pollack, the author of Spring.NET, is the project lead for the Spring Data project.

The Spring Data Neo4j project, as part of the Spring Data initiative, aims to simplify development with the Neo4j graph database. Like JPA, it uses annotations on simple POJO domain objects. The annotations activate one of the supported mapping approaches, either the simple mapping or the advanced AspectJ mapping. Both use the annotation and reflection metadata for mapping the POJO entities and their fields to nodes, relationships, and properties in the graph database.

Spring Data Neo4j allows, at any time, to drop down to the Neo4j-API level, see Introduction to Neo4j to execute functionality with the highest performance possible.

For integration of Neo4j and Grails/GORM please refer to the Neo4j grails plugin. There are also Python bindings as well as community-provided bindings to use Neo4j in embedded or REST mode.

Reference Documentation Overview

The explanation of Spring Data Neo4j's programming model starts with some underlying details. The basic internal workings of the two mapping modes are explained in the initial chapter. Object Graph Mapping covers the simple mapping and Advanced Mapping with AspectJ contains details about the advanced mapping. It also explains some of the common issues around AspectJ tooling with the current IDEs.

To get started with a simple application, you need only your domain model and 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.

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

Relationships between entities are first class citizens in a graph database and therefore worth a separate chapter ([reference_programming_model:relationships]) describing their usage in Spring Data Neo4j.

Indexing operations are useful for finding individual nodes and relationships in a graph. They can be

used to start graph operations or to be processed in your application. Indexing in the plain Neo4j API is a bit more involved. Spring Data Neo4j maintains automatic indexes per entity class, with `@Indexed` annotations on relevant fields. (Indexing)

Being a Spring Data library, Spring Data Neo4j offers a comprehensive Neo4j-Template (Neo4jTemplate) for interacting with the mapped entities and the Neo4j graph database. The operations provided by Spring Data Neo4j - Repositories per mapped entity class are based on the API offered by the Neo4j-Template. It also provides the operations of the Neo4j Core API in a more convenient way. Especially the querying (Indexes, Cypher and Traversals) and result conversion facilities allow writing very concise code.

Spring Data Commons provides a very powerful repository infrastructure that is also leveraged in Spring Data Neo4j. Those repositories consist only 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. At least for typical CRUD, index- and query-operations that is very convenient. The repositories are extensible by annotated, named or derived finder methods. For custom implementations of repository methods you are free to add your own code. (CRUD with repositories).

To be able to leverage the schema-free nature of Neo4j it is possible to project any entity to any other entity type. That is useful as long as they share some properties (or relationships). The entities don't have to share any super-types or hierarchies. How that works is explained here: Projecting entities.

Spring Data Neo4j also allows you to integrate with the powerful geospatial graph library Neo4j-Spatial that offers full support for working with any kind of geo-data. Spring Data Neo4j repositories expose a couple of those operations via bounding-box and near-location searches. Geospatial Queries.

Using computed fields that are dynamically backed by graph operations is a bit more involved. First you should know about traversals and Cypher queries. Those are explained in Introduction to Neo4j. Then you can start using virtual, computed fields in your entities Projecting entities .

If you like the ActiveRecord approach that uses persistence methods mixed into the domain classes, you will want to look at the description of the additional entity methods (see Active Record Methods for Advanced Mapping Mode) that are added to your domain objects by Spring Data Neo4j Aspects. Those allow you to manage the entity lifecycle as well as to connect entities. Those methods also provide the means to execute the mentioned graph operations with your entity as a starting point.

Neo4j is a fully ACID, enterprise grade database. It uses Java transactions, and internally a 2-phase commit protocol, to guarantee the safety of your data. The implications of that are described in the chapter around transactions. (Transactions)

The need of an active transaction for mutating the state of nodes or relationships implies that direct changes to the graph are only possible in a transactional context. Unfortunately many higher level application layers don't want to care about transactions and the open-session-in-view pattern is not widely used. Therefore Spring Data Neo4j's advanced mappings introduced an entity lifecyle and added support for detached entities which can be used for temporary domain objects that are not

intended to be stored in the graph or which will be attached to the graph only later. (Detached node entities in advanced mapping mode)

For the simple mapping this is not neccessary as domain objects are detached by default and have to be explicitly reattached to the graph to store the changes.

Unlike Neo4j which is a schema free database, Spring Data Neo4j works on Java domain objects. So it needs to store the type information in the graph to be able to reconstruct the entities when just nodes are retrieved. To achieve that it employs type-representation-strategies which are described in a separate chapter. (see Entity type representation)

Spring Data Neo4j offers basic support for bean property validation (JSR-303). Annotations from that JSR are recognized and evaluated whenever a property is set, or when a previously detached entity is persisted to the graph. (see Bean validation (JSR-303))

Unfortunately the setup of Spring Data Neo4j advanced mapping mode is more involved than we'd like. That is partly due to the Maven setup and dependencies for AspectJ, which can be alleviated by using different build systems like Gradle or Ant/Ivy. The Spring configuration itself boils down to two lines of `<spring-neo4j>` namespace setup. (see Environment setup)

In a polyglot persistence context Spring Data Neo4j can also be used in a JPA environment to add graph features to your JPA entities. In the Cross-store persistence the slightly different behavior and setup of a Graph-JPA interaction are described.

The provided samples, which are also publicly hosted on Github, are explained in Sample code.

The performance implications of using Spring Data Neo4j are detailed in Performance considerations. This chapter also discusses which use cases should not be handled with Spring Data Neo4j.

As AspectJ might not be well known to everyone, some of the core concepts of the aspect oriented, advanced mapping mode for Java are explained in AspectJ details.

How to consume the REST-API of a Neo4j-Server is the topic of Neo4j Server. But Spring Data Neo4j can also be used to create custom Extensions for the Neo4j Server which would serve domain model abstractions to a suitable front-end. So instead of talking low level primitives to a database, the front-end or web-app would communicate via a domain level protocol with endpoints implemented in Jersey and Spring Data Neo4j.

NOTE	Please be aware that the advanced mapping mode of Spring Data Neo4j is based on AspectJ and uses some advanced features of that toolset. See the section on AspectJ (Advanced Mapping with AspectJ) for details if you run into any problems.

Chapter 21. Introduction to Neo4j

21.1. What is a graph database?

A graph database is a storage engine that is specialized in storing and retrieving vast networks of data. It efficiently stores nodes and relationships and allows high performance traversal of those structures. Properties can be added to nodes and relationships.

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 modeling approaches, the relationships between things are reduced to a single link without identity and attributes. Graph databases allow to keep the rich relationships that originate from the domain, equally well-represented in the database without resorting to also modeling the relationships as "things". There is very little "impedance mismatch" when putting real-life domains into a graph database.

21.2. About Neo4j

Neo4j is a NOSQL graph database. It is a fully transactional database (ACID) that stores data structured as graphs. A graph consists of nodes, connected by relationships. Inspired by the structure of the human mind, it allows for high query performance on complex data, while remaining intuitive and simple for the developer.

Neo4j has been in commercial development for 10 years and in production for over 7 years. Most importantly it has a helpful 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 optimized 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 traversal framework and query languages for traversing the graph.
- can be deployed as a standalone server or an embedded database with a very small distribution footprint.
- has a core Java API

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

21.3. GraphDatabaseService

The API of `org.neo4j.graphdb.GraphDatabaseService` provides access to the storage engine. Its features include creating and retrieving nodes and relationships, managing indexes (via the IndexManager), database life cycle callbacks, transaction management, and more.

The `EmbeddedGraphDatabase` is an implementation of GraphDatabaseService that is used to embed Neo4j in a Java application. This implementation is used so as to provide the highest and tightest integration with the database. Besides the embedded mode, the Neo4j server provides access to the graph database via an HTTP-based REST API.

21.4. Creating nodes and relationships

Using the API of GraphDatabaseService, it is easy to create nodes and relate them to each other. Relationships are typed and both nodes and relationships can have properties. Property values can be primitive Java types and Strings, or arrays of both. As of Neo4j 2.0, any operation on a node or relationship (creation, modification or simply reading) must happen within a transaction.

Example 45. Neo4j usage

```
GraphDatabaseService graphDb = new GraphDatabaseFactory().newEmbeddedDatabase(
"helloworld");

try (Transaction tx = graphDb.beginTx()) {
Node firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
Node secondNode = graphDb.createNode();
secondNode.setProperty( "message", "world!" );

Relationship relationship = firstNode.createRelationshipTo( secondNode,
DynamicRelationshipType.of("KNOWS") );
relationship.setProperty( "message", "brave Neo4j" );
tx.success();
}
```

21.5. Graph traversal

Getting a single node or relationship and examining it is not the main use case of a graph database. Fast graph traversal of complex, interconnected data and application of graph algorithms are. Neo4j provides a DSL for defining `TraversalDescription`s that can then be applied to a start node and will produce a lazy `java.lang.Iterable` result of nodes and/or relationships.

Example 46. Traversal usage

```
TraversalDescription traversalDescription = Traversal.description()
   .depthFirst()
   .relationships(KNOWS)
   .relationships(LIKES, Direction.INCOMING)
   .evaluator(Evaluators.toDepth(5));
for (Path position : traversalDescription.traverse(myStartNode)) {
   System.out.println("Path from start node to current position is " + position);
}
```

21.6. Indexing

The best way for retrieving start nodes for traversals and queries is by using Neo4j's integrated index facilities. NOTE: As of SDN 3.0 , schema based indexes (i.e. indexes based on labels) are the default, however the legacy indexing functionality still remains, as there is some functionality (for example full text searches, range searches) which is not possible/ available yet. It should be noted that legacy based indexes are deprecated in 3.0 and the intention is to eventually remove it completely as and when schema based indexes/functionality is fully able to support existing functionality.

```
The `GraphDatabaseService` still provides access to the legacy `IndexManager` which in
turn provides named indexes for nodes and relationships. Both can be indexed with
property names and values. Retrieval is done with query methods on indexes, returning an
`IndexHits` iterator.
```

Spring Data Neo4j provides automatic indexing via the `@Indexed` annotation, defaulting to make use of schema based indexes (aka labels), eliminating the need for manual index management.

Example 47. Legacy Index usage

```
IndexManager indexManager = graphDb.index();
Index<Node> nodeIndex = indexManager.forNodes("a-node-index");
Node node = ...;
try (Transaction tx = graphDb.beginTx()) {
    nodeIndex.add(node, "property","value");
    tx.success();
}
try (Transaction tx = graphDb.beginTx()) {
    for (Node foundNode : nodeIndex.get("property","value")) {
        // found node
    }
    tx.success();
}
```

21.7. Querying the Graph with Cypher

Neo4j provides a graph query language called "Cypher" which draws from many sources. It resembles SQL but with an iconic representation of patterns in the graph (concepts drawn from SPARQL). The Cypher execution engine was written in Scala to leverage the high expressiveness for lazy sequence operations of the language and the parser combinator library. A screencast explaining the possibilities in detail can be found on the Neo4j video site.

As of Neo4 2.0, Cypher queries typically begin with a `match` clause, although the optional `start` clause (only really needed when using legacy indexes) is also still supported. The `match` clause can be used to provide a way to pattern match against a starting set of nodes, via their IDs or label based index lookup, with the legacy `start` clause providing similar functionality. These starting patterns or start nodes, are then related to other nodes via additional `match` clauses. Start and/or 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. Aggregation also happens in the return clause by using aggregation functions on some of the values. Sorting can happen in the `order by` clause and the `skip` and `limit` parts restrict the result set to a certain window.

Cypher can be executed on an embedded graph database using an `ExecutionEngine` and `CypherParser`. This is encapsulated in Spring Data Neo4j with `CypherQueryEngine`. The Neo4j-REST-Server comes with a Cypher-Plugin that is accessible remotely and is available in the Spring Data Neo4j REST-Binding.

Example 48. Cypher Examples on the Cineasts.net Dataset

```
// ------------------------------------------------------
//                schema based (Label) examples
// ------------------------------------------------------
//               TODO - once code has been updated

// ------------------------------------------------------
//                Legacy index based examples
// ------------------------------------------------------

// Actors who played a Matrix movie :
start movie=node:Movie("title:Matrix*") match movie<-[:ACTS_IN]-actor
    return actor.name, actor.birthplace?

// User-Ratings:
start user=node:User(login='micha') match user-[r:RATED]->movie where r.stars > 3
    return movie.title, r.stars, r.comment

// Mutual Friend recommendations:
start user=node:Micha(login='micha') match user-[:FRIEND]-friend-[r:RATED]->movie
where r.stars > 3
    return friend.name, movie.title, r.stars, r.comment?

// Movie suggestions based on a movie:
start movie=node:Movie(id='13') match (movie)<-[:ACTS_IN]-()-[:ACTS_IN]->(suggestion)
    return suggestion.title, count(*) order by count(*) desc limit 5

// Co-Actors, sorted by count and name of Lucy Liu
start lucy=node(1000) match lucy-[:ACTS_IN]->movie<-[:ACTS_IN]-co_actor
    return count(*), co_actor.name order by count(*) desc,co_actor.name limit 20

// Recommendations including counts, grouping and sorting
start user=node:User(login='micha') match user-[:FRIEND]-()-[r:RATED]->movie
    return movie.title, AVG(r.stars), count(*) order by AVG(r.stars) desc, count(*)
desc
```

Chapter 22. Programming model

This chapter covers the fundamentals of the programming model behind Spring Data Neo4j. It discusses the simple and advanced mapping modes, the annotations provided by Spring Data Neo4j and how to use them. Examples for this section are taken from the "IMDB" project of Spring Data Neo4j examples.

22.1. Object Graph Mapping

Up until recently Spring Data Neo4j supported only the more advanced and flexible AspectJ based mapping approach, see Advanced Mapping with AspectJ. Feedback about issues with the AspectJ tooling and other implications persuaded us to add a simpler mapping (see Simple Object Graph Mapping) to Spring Data Neo4j. Both versions work with the same annotations and provide similar API's, but differ in behaviour.

Reflection and Annotation-based metadata is collected about persistent entities in the `Neo4jMappingContext` which provides it to any part of the library. The information is stored in `Neo4jPersistentEntity` instances which hold all the `Neo4jPersistentProperty's of the type. Each entity can be checked to determine whether it represents a Node or a Relationship. Properties declare detailed data about their indexing and relationship information as well as type information that also covers nested generic types. With all that information available it is simple to select the appropriate strategy for mapping each entity and field to elements, relationships and properties of the graph.

The main difference is in the way of accessing the graph. In the simple mapping the required information is copied into the entity on load and only stored back when an explicit save operation occurs. In the advanced mapping (AspectJ-enhanced) approach a node or relationship is attached via an additional field to the entity and all read- and write-operations (inside of Transactions) happen through that.

For the simple mapping mode, declaration of fetch strategies for related entities is necessary to avoid loading the whole graph eagerly into memory. The initial approach uses just a simple `@Fetch` annotations on relationship properties. The resulting `MappingPolicy` is provided to the infrastructure methods to ensure the correct loading behaviour. Both, `Neo4jPersistentEntiy` and `Neo4jPersistentProperty` can be queried for the `MappingPolicy`.

Otherwise the two approaches share much of the infrastructure. E.g. for creating new entity instances from type information store in the graph (Entity type representation), the infrastructure for mapping individual fields to graph properties and relationships and everything related to indexing and querying. A certain part of that is also exposed via the Neo4jTemplate for direct use.

22.2. Advanced Mapping with AspectJ

Behind the scenes, Spring Data Neo4j leverages AspectJ aspects to modify the behavior of annotated POJO entities (see AspectJ details). Each node entity is backed by a graph node that holds its properties

and relationships to other entities. AspectJ is used for intercepting field access, so that Spring Data Neo4j can retrieve the appropriate information from the entity's backing node or relationship.

The aspect introduces an internal field (`entityState`) and some public methods (see Active Record Methods for Advanced Mapping Mode) to the entities, for instance `entity.getPersistentState()` and `entity.relateTo`. It also introduces some methods for graph operations that start at the current entity. Introduced methods for `equals()` and `hashCode()` use the underlying node or relationship. Please take the introduced field into account when serializing your entities and exclude it from the serialization process.

Spring Data Neo4j internally uses an abstraction called `EntityState` that the field access and instantiation advices of the aspect delegate to. This way, the aspect code is kept to a minimum, focusing mainly on the pointcuts and delegation. The `EntityState` then uses a number of `FieldAccessorFactories` to create a `FieldAccessor` instance per field that does the specific handling needed for the concrete field type. There is some caching involved as well, so it handles repeated instantiation efficiently.

To use the advanced, AspectJ based mapping, please add `spring-data-neo4j-aspects` as a dependency and set up the AspectJ integration in Maven or other build tools as explained in Environment setup. Some hints for your IDE setup are described below.

22.2.1. AspectJ IDE support

As Spring Data Neo4j uses some advanced features of AspectJ, users may experience issues with their IDE reporting errors where in fact there are none. Features that might be reported wrongfully include: introduction of methods to interfaces, declaration of additional interfaces for annotated classes, and generified introduced methods.

IDEs not providing full AspectJ support might mark parts of your code as having errors. You should rely on your build-system and tests to verify the correctness of the code. You might also have your Entities (or their interfaces) implement the `NodeBacked` and `RelationshipBacked` interfaces directly to benefit from completion support and error checking.

Eclipse and STS support AspectJ via the AJDT plugin which can be installed from the update-site listed at http://www.eclipse.org/ajdt/downloads/ (it might be necessary to use the latest development snapshot of the plugin). The current version that does not show incorrect errors is AspectJ 1.6.12 (included in STS 2.8.0), previous versions are reported to mislead the user. Note that AJDT (as of September 2012) requires projects to be rebuild after Eclipse is started to fully support all advanced features.

NOTE

There might be some issues with the eclipse maven plugin not adding AspectJ files correctly to the build path. If you encounter issues, please try the following: Try editing the build path to `include **/*.aj` for the spring-data-neo4j-aspects project. You can do this by selecting "Build Path Configure Build Path ..." from the Package Explorer. Then for the `spring-data-neo4j-aspects/src/main/java` add `*/.aj` to the Included path. When importing a Spring Data Neo4j project into Eclipse with m2e, please make sure the AspectJ Configurator is installed from the following update-site: http://dist.springsource.org/release/AJDT/configurator

The AspectJ support in IntelliJ IDEA lacks some of the features. JetBrains is working on improving the situation in their upcoming 11 release of their popular IDE. Their latest work is available under their early access program (EAP). Building the project with the AspectJ compiler `ajc` works in IDEA (Options Compiler Java Compiler should show ajc). Make sure to give the compiler at least 512 MB of RAM.

22.3. Simple Object Graph Mapping

In addition to the advanced object graph mapping using AspectJ, Spring Data Neo4j also supports a simpler mode that converts graph data into domain objects and vice versa. It does not require any additional set up and should work out of the box. The simple mapping approach uses the same annotations ([null]) as the advanced mapping to declare mapping meta-information.

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

It uses the available meta-information about the persistent entity to iterate over its properties and relationships, fetching their data from the graph while doing so. It also executes computed fields and stores the resulting values in the properties.

We try to avoid loading the whole graph into memory by not following relationships eagerly. A dedicated `@Fetch` annotation controls instead if related entities are loaded or not. Whenever an entity is not fully loaded, then only its id is stored. Those entities or collections of entities can then later be loaded explicitly using the `template.fetch()` operation.

The additional fetch information is stored in a `MappingPolicy` which can be retrieved via the `Neo4jTemplate` for classes. Both `Neo4jPersistentEntitity` as well as `Neo4jPersistentProperty` provide access to that information on their scope.

NOTE

Please note that if you have two collections in an entity pointing to the same relationship and one of them has data and the other is empty due to the nature of persisting it, one will override the other in the graph so that you might end up with no data. If you want a relationship-collection to be ignored on save set it to null.

Example 49. Examples for loading entities from the graph

```
@Autowired Neo4jOperations template;

@NodeEntity class Person {
String name;
@Fetch Person boss;
Person spouse;

@RelatedTo(type = "FRIEND", direction = BOTH)
@Fetch Set<Person> friends;
}
Person person = template.findOne(personId);
assertNotNull(person.getBoss().getName());

assertNotNull(person.getSpouse().getId());
assertNull(person.getSpouse().getName());

template.fetch(person.getSpouse());
assertNotNull(person.getSpouse().getName());

assertEquals(10,person.getFriends().size());
assertNotNull(firstFriend.getName());
```

NOTE Both the simple mapping approach as well as the fetch strategies (`MappingPolicy`) debuted in Spring Data Neo4j 2.0. So there might be rough edges and there are certainly many areas for improvement and extension. We look forward to your feedback on this topic.

As we tried to encapsulate each aspect of the mapping process into a separate class the resulting fabric of responsibilities is quite intricate. All of them are set up in the `MappingInfrastructure` that is part of the `Neo4jTemplate` setup.

22.4. Defining node entities

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

22.4.1. @NodeEntity: The basic building block

The `@NodeEntity` annotation is used to turn a POJO class into an entity backed by a node in the graph database. 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. If the `useShortNames` attribute is set to false, the property and relationship names will have the class name of the entity prepended.

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

If the `partial` attribute is set to true, this entity takes part in a cross-store setting, where the entity lives in both the graph database and a JPA data source. See Cross-store persistence for more information.

Entity fields can be annotated with `@GraphProperty`, `@RelatedTo`, `@RelatedToVia`, `@Indexed`, `@GraphId`, `@Query` and `@GraphTraversal`.

Example 50. Simplest node entity

```
@NodeEntity
public class Movie {
    String title;
}
```

22.4.2. @GraphId: Neo4j -id field

For the simple mapping this is a required field which must be of type `Long`. It is used by Spring Data Neo4j to store the node or relationship-id to re-connect the entity to the graph.

NOTE	It must not be a primitive type because then the "non-attached" case can not be represented as the default value 0 would point to the reference node. Please make also sure that an `equals()` and `hashCode()` method have to be provided which take the `id` field into account (and also handle the "non-attached", null case).

For the advanced mapping such a field is optional. Only if the underlying id has to be accessed, it is needed.

Entity Equality

Entity equality can be a grey area, and it is debatable whether natural keys or database ids best describe equality, there is the issue of versioning over time, etc. For Spring Data Neo4j we have adopted the convention that database-issued ids are the basis for equality, and that has some consequences:

1. Before you attach an entity to the database, i.e. before the entity has had its id-field populated, we suggest you rely on object identity for comparisons
2. Once an entity is attached, we suggest you rely solely on the id-field for equality
3. When you attach an entity, its hashcode changes - because you keep equals and hashcode consistent and rely on the database ID, and because Spring Data Neo4j populates the database ID on save

That causes problems if you had inserted the newly created entity into a hash-based collection before

saving. While that can be worked around, we strongly advise you adopt a convention of not working with un-attached entities, to keep your code simple. This is best illustrated in code.

Example 51. Entity using id-field for equality and attaching new entity immediately

```
@NodeEntity
public class Studio {
    @GraphId
    Long id

    String name;

    public boolean equals(Object other) {
        if (this == other) return true;

        if (id == null) return false;

        if (! (other instanceof Studio)) return false;

        return id.equals(((Studio) other).id);
    }

    public int hashCode() {
        return id == null ? System.identityHashCode(this) : id.hashCode();
    }
}

...
Set<Studio> studios = new HashSet<Studio>();
Studio studio = studioRepository.save(new Studio("Ghibli"));
studios.add(studio);
Studio sameStudio = studioRepository.findOne(studio.id);
assertThat(studio, is(equalTo(sameStudio));
assertThat(studios.contains(sameStudio), is(true);
assertThat(studios.remove(sameStudio), is(true);
```

A work-around for the problem of un-attached entities having their hashcode change when they get saved is to cache the hashcode. The hashcode will change next time you load the entity, but at least if you have the entity sitting in a collection, you will still be able to find it:

Example 52. Caching hashcode

```
@NodeEntity
public class Studio {
    @GraphId
    Long id

    String name;

    transient private Integer hash;

    public boolean equals(Object other) {
        if (this == other) return true;

        if (id == null) return false;

        if (! (other instanceof Studio)) return false;

        return id.equals(((Studio) other).id);
    }

    public int hashCode() {
        if (hash == null) hash = id == null ? System.identityHashCode(this) : id
.hashCode();

        return hash.hashCode();
    }
}

...
Set<Studio> studios = new HashSet<Studio>();
Studio studio = new Studio("Ghibli")
studios.add(studio);
studioRepository.save(studio);
assertThat(studios.contains(studio), is(true);
assertThat(studios.remove(studio), is(true);
Studio sameStudio = studioRepository.findOne(studio.id);
assertThat(studio, is(equalTo(sameStudio));
assertThat(studio.hashCode(), is(not(equalTo(sameStudio.hashCode()))));
```

NOTE Remember, transient fields are **not** saved.

22.4.3. @GraphProperty: Optional annotation for property fields

It is not necessary to annotate property fields, as they are persisted by default; 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 a custom conversion factory that comes with converters for `Enum`s and `Date`s. Transient fields are not persisted.

Collections of of primitive or convertible values are stored as well. They are converted to arrays of their type or strings respectively.

This annotation is typically used with cross-store persistence. When a node entity is configured as partial, then all fields that should be persisted to the graph must be explicitly annotated with `@GraphProperty`.

`@GraphProperty` can specify default values for properties that are not in the graph. Default values are specified as String representations and will be converted to the correct target type using the existing conversion facilities. For example `@GraphProperty(defaultValue="20") Integer age`.

It is also possible to declare the type that should be used for the storage inside of Neo4j. For instance if a `Date` property should be stored as an Long value instead of the default String, the annotation would look like `@GraphProperty(propertyType = Long.class)` For the actual mapping of the Field-Type to the Neo4j-Property type there has to be a Converter registered in the Spring-Config.

Finally, node property names can also be explicitly assigned by using the `propertyName` attribute. For example `@GraphProperty(propertyName="last_name") String lastName`. The node property name defaults to the field name when not specified.

22.4.4. @Indexed: Making entities searchable by field value

The @Indexed annotation can be declared on fields that are intended to be indexed by the Neo4j indexing facilities. The resulting index can be used to later retrieve nodes or relationships that contain a certain property value, e.g. a name. Often an index is used to establish the start node for a traversal. Indexes are accessed by a repository for a particular node or relationship entity type. See Indexing and CRUD with repositories for more information.

22.4.5. @Query: fields as query result views

The `@Query` annotation leverages the delegation infrastructure supported by Spring Data Neo4j. It provides dynamic fields which, when accessed, return the values selected by the provided query language expression. The provided query must contain a placeholder named `{self}` for the the current entity. For instance the query `start n=node({self}) match n-[:FRIEND] friend return friend`. Graph queries can return variable number of entities. That's why annotation can be put onto fields with a single value, a subclass of Iterable of a concrete type or an Iterable of `Map<String,Object>`. Additional parameters are taken from the params attribute of the `@Query` annotation. These parameter tuples form key-value pairs that are provided to the query at execution time.

Example 53. @Graph on a node entity field

```
@NodeEntity
public class Group {
    @Query(value = "start n=node({self}) match (n)-[r]->(friend) where r.type =
{relType} return friend",
                params = {"relType", "FRIEND"})
    private Iterable<Person> friends;
}
```

NOTE Please note that this annotation can also be used on repository methods. (CRUD with repositories)

22.4.6. @GraphTraversal: fields as traversal result views

The `@GraphTraversal` annotation also leverages the delegation infrastructure supported by Spring Data aspects. It provides dynamic fields which, when accessed, return an `Iterable` of node or relationship entities that are the result of a traversal starting at the entity containing the field. The `TraversalDescription` used for this is created by the `FieldTraversalDescriptionBuilder` class defined by the `traversal` attribute. The class of the resulting node entities must be provided with the `elementClass` attribute.

Example 54. @GraphTraversal from a node entity

```
@NodeEntity
public class Group {
    @GraphTraversal(traversal = PeopleTraversalBuilder.class,
            elementClass = Person.class, params = "persons")
    private Iterable<Person> people;

    private static class PeopleTraversalBuilder implements
FieldTraversalDescriptionBuilder {
        @Override
        public TraversalDescription build(NodeBacked start, Field field, String...
params) {
            return new TraversalDescriptionImpl()
                    .relationships(DynamicRelationshipType.withName(params[0]))
                    .filter(Traversal.returnAllButStartNode());
        }
    }
}
```

22.5. Relating node entities

Since relationships are first-class citizens in Neo4j, associations between node entities are represented by relationships. In general, relationships are categorized 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.

NOTE	As of Neo4j 1.4.M03, circular references are allowed. Spring Data Neo4j reflects this accordingly.

22.5.1. @RelatedTo: Connecting node entities

Every field of a node 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 field pointing to another node entity (1:1). In this case, the field 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 field, a relationship is created when the entity is persisted. If the field is set to `null`, the relationship is removed.

Example 55. Single relationship field

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

It is also possible to have fields that reference a set of node entities (1:N). These fields come in two forms, modifiable or read-only. Modifiable fields are of the type `Set<T>`, and read-only fields are `Iterable<T>`, where T is a @NodeEntity-annotated class.

Example 56. Node entity with relationships

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

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

For the simple mapping, the automatic transitive loading of related entities depends on declaration of `@Fetch` at the property. Otherwise the related node or relationship entities will just be initialized with their id for later loading.

When using the advanced mapping, Fields referencing other entities should not be manually initialized, as they are managed by Spring Data Neo4j Aspects under the hood. 1:N fields can be accessed immediately, and Spring Data Neo4j will provide a `Set` representing the relationships.

If this `Set` of related entities is modified, the changes are reflected in the graph, relationships are added, removed or updated accordingly.

NOTE Spring Data Neo4j ensures by default that there is only one relationship of a given type between any two given entities. This can be circumvented by using the `createRelationshipBetween()` method with the `allowDuplicates` parameter on repositories or entities.

NOTE Before an entity has been persisted for the first time, it will not have its state managed by Spring Data Neo4j. For example, given the Actor class defined above, if `actor.movies` was accessed in a non-persisted entity, it would return `null`, whereas if it was accessed in a persisted entity, it would return an empty managed set.

When an Interface is used as target type for the `Set` and/or as `elementClass` it should be marked as `@NodeEntity` too.

By setting direction to `BOTH`, relationships are created in the outgoing direction, but when the 1:N field is read, it will include relationships in both directions. A cardinality of M:N is not necessary because relationships can be navigated in both directions.

In the advanced mapping mode, the relationships can also be accessed by using the methods `entity.getRelationshipBetween(target, type)` and `entity.relateTo(target, type)` available on each NodeEntity. These methods find and create Neo4j relationships. It is also possible to manually remove relationships by using `entity.removeRelationshipTo(target, type)`. Using these methods is significantly faster than adding/removing from the collection of relationships as it doesn't have to re-synchronize a whole set of relationships with the graph.

Methods of the same semantics exist in the repositories to be used in the simple mapping mode.

NOTE Other collection types than `Set` are not supported so far, also currently NO `Map<RelationshipType,Set<NodeBacked>>`.

22.5.2. @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. As described above, fields annotated with `@RelatedTo` provide a way to only link node entities via relationships, but it provides no way of accessing the relationships themselves.

Relationship entities can be accessed via by @RelatedToVia-annotated ([reference_programming_model:relationships:relatedtovia]) fields or methods like `entity.getRelationshipTo()` or `template|repository.getRelationship(s)Between()`.

Relationship entities either be instantiated directly and set or added to `@RelatedToVia`-annotated fields or created by the introduced `entity.relateTo()`, `template|repository.createRelationshipBetween()` methods (see alos Active Record Methods for Advanced Mapping Mode)

Fields in relationship entities are, similarly to node entities, 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 read-only access to the corresponding endpoint, depending on the chosen annotation.

For the relationship-type a `String` or `RelationshipType` field annotated with `@RelationshipType` is available. When Relationship-Entities are instantiated directly, the relationship type has to be provided either in this annotated field or as part of the `@RelationshipEntity` annotation.

Example 57. Relationship entity (in advanced mapping)

```
@NodeEntity
public class Actor {
    public Role playedIn(Movie movie, String title) {
        return relateTo(movie, Role.class, "ACTS_IN");
    }
}

@RelationshipEntity
public class Role {
    String title;

    @StartNode private Actor actor;
    @EndNode private Movie movie;
}
```

22.5.3. @RelatedToVia: Accessing relationship entities

To provide easy programmatic access to the richer relationship entities of the data model, the annotation `@RelatedToVia` can be added on fields of type `Iterable<T>` or `Set<T>` or T, where T is a `@RelationshipEntity`-annotated class. These fields provide access to relationship entities.

Example 58. Relationship entity (in simple mapping)

```
@NodeEntity
public class Actor {
    @RelatedToVia
    @Set<Role> roles=new HashSet<Role>();
    public Role playedIn(Movie movie, String title) {
        Role role=new Role(this,movie,title);
        roles.add(role);
        return role;
    }
    @RelatedToVia(type="FRIEND_OF", direction=Direction.INCOMING)
    Friendship bestFriend;
}

@RelationshipEntity(type = "ACTS_IN")
public class Role {
    String title;

    @StartNode private Actor actor;
    @EndNode private Movie movie;
}

@RelationshipEntity
public class Friendship {
    Date since;

    @StartNode private Actor actor;
    @EndNode private Person buddy;
}
```

22.5.4. Relationship Type Precedence

In the example above we show how to specify a default relationship type, and how to provide the relationship type using an annotation property. Here is an example of using the @RelationshipType annotation on a member variable on the relationship entity; we call this dynamic relationship type.

Example 59. Dynamic Relationship Type (simple mapping)

```
@RelationshipEntity(type = "colleague")
public class Acquaintance {
      @StartNode private Actor actor;
      @EndNode private Person acquaintance;
      @RelationshipType private String connection;

      public Acquaintance(Actor actor, Person acquaintance, String connection) {
          ...

Actor frankSinatra = ...
Person carloGambino = ...
new Acquaintance(frankSinatra, carloGambino, "its_complicated")
```

NOTE Because dynamic type information is, well, dynamic, it is generally not possible to read the mapping backwards using SDN. The relationship still exists, but SDN cannot help you access it because it does not know what type you gave it. Also, for this reason, we require you to specify a default relationship type, so that we can at least attempt the reverse mapping.

Should you happen to provide conflicting relationship types, we have established the following precedence, in priority order:

1. Dynamic
2. Annotation-provided
3. Default

22.5.5. 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:

Example 60. Clashing Relationship Types

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

    @RelatedTo(type="OWNS")
    Pet pet;
...
```

It is clear how we can map these relationships: by looking at the type of the end node. To enable this, we have introduced an boolean annotation parameter `enforceTargetType`, which is disabled by default. Our example now reads:

Example 61. Discriminating Relationship Types Using End Node Type

```
@NodeEntity
class Person {
    @RelatedTo(type="OWNS", enforceTargetType=true)
    Car car;

    @RelatedTo(type="OWNS", enforceTargetType=true)
    Pet pet;
...
```

The example easily generalises to collections too of course, but there are a few note-worthy rules and corner cases:

- You need to annotate **all** clashing relationships.
- You can't have two fields, two collections, or a field and a collection, with the same relationship type and identical end node types. SDN does not store metadata about the origin of a relationship. So when saving the entity, the first field or collection would be overwritten by the second, with the processing order being non-deterministic.
- You **can** have clashing relation ship types when end nodes share a supertype.
- A variation on the above, you **cannot** have two fields or two collections with the same relationship type and substitutable end node types.
- You **can** however have a field and a collection where end node types inherit from each other.

22.6. Indexing

Indexing is used in Neo4j to quickly find nodes and relationships to start graph operations from. Either for manually traversing the graph, using the traversal framework, cypher queries or for "global" 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 and Spring Data Neo4j 3.0. 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.

22.6.1. Schema (Label based) indexes

Since Neo4j version 2.0 indexes and unique constraints based on labels and properties are supported throughout the API including cypher. For properties of entities annotated with `@Indexed`, this defaults to using the schema based strategy, and an appropriate schema index is created. For `@Indexed(unique=true)` a constraint is created.

Those indexes will be automatically used by cypher queries that are generated for the derived finders and are available for custom queries.

22.6.2. Index Creation

Indexes are created and updated upfront, when Spring Data Neo4j scans your entities and detects indexed fields. It will use an instance of `org.springframework.data.neo4j.support.mapping.EntityIndexCreator to create both schema based and legacy indexes.

For certain setups, e.g. when your system is already set-up with all the indexes or when you run in an HA-cluster against a slave, you can disable this behavior by configuring `<neo4j:config create-index="false" />` or calling `Neo4jConfiguration.setCreateIndex(false)`. The default value is "true".

22.6.3. Legacy indexes

If you would like to force a property on an entity to rather use the legacy index (instead of the default schema based index), then you will need to explicitly specify the type as either `@Indexed(indexType = IndexType.SIMPLE)` or `@Indexed(indexType = IndexType.FULLTEXT)`

The Neo4j graph database employs different index providers for legacy exact (SIMPLE) lookups and fulltext searches. Lucene is the default index provider implementation. Each named index is configured to be fulltext or exact. There is also a spatial index provider for geo-searches.

22.6.4. Exact and numeric index

Prior to Neo4j 2.0, when using the standard Neo4j API, nodes and relationships had to be manually indexed with key-value pairs, typically being the property name and value. With the introduction of schemas and labels, indexing now happens automatically for you under the covers. When using Spring Data Neo4j, irrespective of whether you are using the newer schema based indexes or legacy indexes, this task is simplified to just adding an `@Indexed` annotation on entity fields by which the entity should be searchable. This will result in automatic updates of the appropriate index every time an indexed field changes.

Numerical fields are indexed numerically so that they are available for range queries. NOTE: Automatic numerical range queries are not currently supported for schema based numeric indexes.

All other fields are indexed with their string representation. If a numeric field should not be indexed numerically, it is possible to switch it off with `@Indexed(numeric=false)`.

The `@Indexed` annotation also provides the option of using a custom index name (for legacy indexes). The default index name is the simple class name of the entity, so that each class typically gets its own index. It is recommended to not have two entity classes with the same class name, regardless of package.

If a field is declared in a superclass but different indexes for subclasses are needed, the `level` attribute declares what will be used as index. `Level.CLASS` uses the class where the field was declared and `Level.INSTANCE` uses the class that is provided or of the actual entity instance.

The schema based indexes can be queried by using a repository (see CRUD with repositories). The repository is an instance of `org.springframework.data.neo4j.repository.SchemaIndexRepository`. The methods `findBySchemaPropertyValue()` and `findAllBySchemaPropertyValue()` work on the exact indexes and return the first or all matches. Range queries are not supported yet.

The legacy indexes can also be queried by using a repository (see CRUD with repositories). The repository is still an instance of the deprecated `org.springframework.data.neo4j.repository.IndexRepository`. The methods `findByPropertyValue()` and `findAllByPropertyValue()` work on the exact indexes and return the first or all matches. To do range queries, use `findAllByRange()` (please note that currently both values are inclusive).

When providing explicit index names (for legacy indexes) the repository has to extend `NamedIndexRepository`. This adds the shown methods with another signature that take the index name as first parameter.

Example 62. Exact (schema based) indexes

```
@NodeEntity
class Person {
    @Indexed String name;
    @Indexed int age;
}

GraphRepository<Person> graphRepository = template.repositoryFor(Person.class);

// Exact match, in named index
Person mark = graphRepository.findBySchemaPropertyValue("name", "mark");
```

Example 63. Exact (legacy) indexes

```
@NodeEntity
class Person {
    @Indexed(indexName = "people",indexType = IndexType.SIMPLE) String name;
    @Indexed(indexType = IndexType.SIMPLE) int age;
}

GraphRepository<Person> graphRepository = template.repositoryFor(Person.class);

// Exact match, in named index
Person mark = graphRepository.findByPropertyValue("people", "name", "mark");

// Numeric range query, index name inferred automatically
for (Person middleAgedDeveloper : graphRepository.findAllByRange("age", 20, 40)) {
    Developer developer=middleAgedDeveloper.projectTo(Developer.class);
}
```

22.6.5. Fulltext (legacy) indexes

Spring Data Neo4j also supports fulltext indexes - currently still only via the legacy indexes. By default, legacy indexed fields are stored in an exact lookup index. To have them analyzed and prepared for fulltext search, the `@Indexed` annotation has the `type` attribute which can be set to `IndexType.FULLTEXT`. Please note that fulltext indexes require a separate index name as the fulltext configuration is stored in the index itself.

Access to the fulltext index is provided by the `findAllByQuery()` repository method. Wildcards like `*` are allowed. Generally though, the fulltext querying rules of the underlying index provider apply. See the Lucene documentation for more information on this.

Example 64. Fulltext indexing

```
@NodeEntity
class Person {
    @Indexed(indexName = "people-search", indexType=IndexType.FULLTEXT) String name;
}

GraphRepository<Person> graphRepository =
             template.repositoryFor(Person.class);

Person mark = graphRepository.findAllByQuery("people-search", "name", "ma*");
```

22.6.6. Unique indexes

Unique indexing can be applied either via the inbuilt schema (label based) unique constraint for nodes, or, via the legacy `index.putIfAbsent` and `UniqueFactory` code for both nodes and relationships. In Spring Data Neo4j this is done by setting the `unique=true` property on the `@Indexed` annotation. Methods for programmatically getting and/or creating unique entities is available on the `Neo4jTemplate` class, namely `getOrCreateNode` and `getOrCreateRelationship` for legacy indexes, and `merge` for schema based unique entities.

In an entity at most one field can be annotated with `@Indexed(unique=true)` regardless of the index-type used. The uniqueness will be taken into account when creating the entity by reusing an existing entity if that unique key-combination already exists. On saving of the field it will be cross-checked against the schema or legacy index and fail with a DataIntegrityViolationException if the field was changed to an already existing unique value. Null values are no longer allowed for these properties.

NOTE

This works for both Node-Entities as well as Relationship-Entities (legacy indexes only). Relationship-Uniqueness in Neo4j is global so that an existing unique instance of this relationship may connect two completely different nodes and might also have a different type.

Example 65. Unique indexing (Schema Based)

```
// creates or finds a node with the unique label-key-value combination
// and initializes it with the properties given
List labels = getTRSLabels(Person.class);
template.merge("Person", "name", "Michael", map("name","Michael","age",37),labels);

@NodeEntity class Person {
    @Indexed(unique = true) String name;
}

Person mark1 = repository.save(new Person("mark"));
Person mark2 = repository.save(new Person("mark"));

// just one node is created
assertEquals(mark1,mark2);
assertEquals(1, personRepository.count());

Person thomas = repository.save(new Person("thomas"));
thomas.setName("mark");
repository.save(thomas); // fails with a DataIntegrityViolationException
```

Example 66. Unique indexing (Legacy Based)

```
// creates or finds a node with the unique index-key-value combination
// and initializes it with the properties given
List labels = getTRSLabels(Person.class);
template.getOrCreateNode("Person", "name", "Michael", map("name","Michael","age",37)
,labels);

@NodeEntity class Person {
    @Indexed(indexType = IndexType.SIMPLE, unique = true) String name;
}

Person mark1 = repository.save(new Person("mark"));
Person mark2 = repository.save(new Person("mark"));

// just one node is created
assertEquals(mark1,mark2);
assertEquals(1, personRepository.count());

Person thomas = repository.save(new Person("thomas"));
thomas.setName("mark");
repository.save(thomas); // fails with a DataIntegrityViolationException
```

22.6.7. Manual (Legacy) index access

The legacy index for a domain class is also available from `Neo4jTemplate` via the `getIndex()` method. The second parameter is optional and takes the index name if it should not be inferred from the class name. It returns the index implementation that is provided by Neo4j. Note: Manual Legacy index access is deprecated in SDN 3.0

Example 67. Manual index retrieval by type and name

```
@Autowired Neo4jTemplate template;

// Default index
Index<Node> personIndex = template.getIndex(null, Person.class);
personIndex.query(new QueryContext(NumericRangeQuery.new  ntRange("age", 20, 40,
true, true))
                        .sort(new Sort(new SortField("age", SortField.INT, false))));

// Named index
Index<Node> namedPersonIndex = template.getIndex("people",Person.class);
namedPersonIndex.get("name", "Mark");

// Fulltext index
Index<Node> personFulltextIndex = template.getIndex("people-search", Person.class);
personFulltextIndex.query("name", "*cha*");
personFulltextIndex.query("{name:*cha*}");
```

It is also possible to pass in the property name of the entity with an `@Indexed` annotation whose index should be returned.

Example 68. Manual index retrieval by property configuration

```
@Autowired Neo4jTemplate template;

Index<Node> personIndex = template.getIndex(Person.class, "age");
personIndex.query(new QueryContext(NumericRangeQuery.new  ntRange("age", 20, 40,
true, true))
                        .sort(new Sort(new SortField("age", SortField.INT, false))));

// Fulltext index
Index<Node> personFulltextIndex = template.getIndex(Person.class,"name");
personFulltextIndex.query("name", "*cha*");
personFulltextIndex.query("{name:*cha*}");
```

22.6.8. Index queries in Neo4jTemplate

For querying the index, the template offers query methods that take either the exact match parameters or a query object/expression, return the results as `Result` objects which can then be converted and projected further using the result-conversion-dsl (see Neo4jTemplate).

22.6.9. Neo4j Auto Indexes

Neo4j allows to configure (legacy) auto-indexing for certain properties on nodes and relationships. This auto-indexing differs from the approach used in Spring Data Neo4j, because there is only one index across all nodes or 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.

22.6.10. Spatial Indexes

Spring Data Neo4j offers limited support for spatial queries using the `neo4j-spatial` library. See the separate chapter Geospatial Queries for details.

22.7. Neo4jTemplate

The `Neo4jTemplate` offers the convenient API of Spring templates for the Neo4j graph database. The Spring Data Neo4j Object Graph mapping builds upon the core functionality of the template to persist objects to the graph and load them in a variety of ways. The template handles the active mapping mode (Object Graph Mapping) transparently.

Besides methods for creating, storing and deleting entities, nodes and relationships in the graph, `Neo4jTemplate` also offers a wide range of query methods. To reduce the proliferation of query methods a simple result handling DSL was added.

22.7.1. Basic operations

For direct retrieval of nodes and relationships, the `getReferenceNode()`, `getNode()` and `getRelationship()` methods can be used.

There are methods (`createNode()` and `createRelationship()`) for creating nodes and relationships that automatically set provided properties.

Example 69. Neo4j template

```
Neo4jOperations neo = new Neo4jTemplate(graphDatabase);

    Node mark = neo.createNode(map("name", "Mark"));
    Node thomas = neo.createNode(map("name", "Thomas"));

    neo.createRelationshipBetween(mark, thomas, "WORKS_WITH",
                                  map("project", "spring-data"));

    neo.index("devs", thomas, "name", "Thomas");

    assertEquals( "Mark",
        neo.query("start p=node({person}) match p<-[:WORKS_WITH]-other return
other.name",
                  map("person", asList(thomas.getId()))).to(String.class).single());



    // Index lookup
    assertEquals(thomas, neo.lookup("devs", "name", "Thomas").to(Node.class).single(
));

    // Index lookup with Result Converter
    assertEquals("Thomas", neo.lookup("devs", "name", "Thomas")
      .to(String.class, new ResultConverter<PropertyContainer, String>() {
        public String convert(PropertyContainer element, Class<String> type) {
            return (String) element.getProperty("name");
        }
      }).single());
```

22.7.2. Core-Operations

`Neo4jTemplate` provides access to some of the methods of the Neo4j-Core-API directly. So accessing nodes and relationships (`getReferenceNode, getNode, getRelationship, getRelationshipBetween`), creating nodes and relationships (`createNode, createNodeAs, createRelationshipBetween`) and deleting them (`delete, deleteRelationshipBetween`) are supported. It also provides access to the underlying GraphDatabase via `getGraphDatabase`.

22.7.3. Entity-Persistence

`Neo4jTemplate` allows to `save`,`saveOnly`, `find(One/All)`, `count`, `delete` and `projectTo` entities. It provides the stored type information via `getStoredJavaType` and can `fetch` lazy-loaded entities or `load` them altogether.

The `saveOnly` method will not reload the entity from the database, saving a number of requests that are often not necessary. It's also available in the `GraphRepository` interface.

22.7.4. Result

All querying methods of the template return a uniform result type: `Result<T>` which is also an `Iterable<T>`. The query result offers methods of converting each element to a target type `result.to(Type.class)` optionally supplying a `ResultConverter<FROM,TO>` which takes care of custom conversions. By default most query methods can already handle conversions from and to: Paths, Nodes, Relationship and GraphEntities as well as conversions backed by registered ConversionServices. A converted `Result<FROM>` is an `Iterable<TO>`. Results can be limited to a single value using the `result.single() or result.singleOrNull()` methods. It also offers support for a pure callback function using a `Handler<T>`.

22.7.5. Indexing

Adding nodes and relationships to an index is done with the `index()` method.

The `lookup()` methods either take a field/value combination to look for exact matches in the index, or a Lucene query object or string to handle more complex queries. All `lookup()` methods return a `Result<PropertyContainer>` to be used or transformed.

22.7.6. Graph traversal

The traversal methods are at the core of graph operations. The `traverse()` method covers the full traversal operation that takes a `TraversalDescription` (typically built with the `template.getGraphDatabase().traversalDescription()` DSL) and runs it from the given start node. `traverse` returns a `Result<Path>` to be used or transformed.

22.7.7. Cypher Queries

The `Neo4jTemplate` also allows execution of arbitrary Cypher queries. Via the `query` methods the statement and parameter-Map are provided. Cypher Queries return tabular results, so the `Result<Map<String,Object>>` contains the rows which can be either used as they are or converted as needed.

22.7.8. Transactions

The `Neo4jTemplate` provides implicit transactions for some of its methods. For instance `save` uses them. For other modifying operations please provide Spring Transaction management using `@Transactional` or the `TransactionTemplate`.

22.7.9. Neo4j REST Server

If the template is configured to use a `SpringRestGraphDatabase` the operations that would be expensive over the wire, like traversals and querying are executed efficiently on the server side by using the

REST API to forward those calls. All the other template methods require individual network operations.

The REST-batch-mode of the `SpringRestGraphDatabase` is not yet exposed via the template, but it is available via the graph database.

22.7.10. Lifecycle Events

Neo4j Template offers basic lifecycle events via Spring's event mechanism using ApplicationListener and ApplicationEvent. The following hooks are available in the form of types of application event:

- BeforeSaveEvent
- AfterSaveEvent
- BeforeDeleteEvent
- AfterDeleteEvent

The following example demonstrates how to hook into the application lifecycle and register listeners that perform behaviour across types of entities during this life cycle:

Example 70. Auditing Entities and Generating Unique Application-level IDs

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

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

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

Changes made to entities in the before-save event handler are reflected in the stored entity - after-save ones are not.

22.8. CRUD with repositories

The repositories provided by Spring Data Neo4j build on the composable repository infrastructure in Spring Data Commons. They 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 repositories support annotated and named queries for the Neo4j Cypher query-language.

Spring Data Neo4j comes with typed repository implementations that provide methods for locating node and relationship entities. There are several types of basic repository interfaces and implementations. `CRUDRepository` provides basic operations, `IndexRepository` and `NamedIndexRepository` delegate to Neo4j's internal indexing subsystem for queries, and `TraversalRepository` handles Neo4j traversals.

With the `RelationshipOperationsRepository` it is possible to access, create and delete relationships between entitites or nodes. The `SpatialRepository` allows geographic searches (Geospatial Queries)

`GraphRepository` is a convenience repository interface, combining `CRUDRepository`, `IndexRepository`, and `TraversalRepository`. Generally, it has all the desired repository methods. If other operations are required then the additional repository interfaces should be added to the individual interface declaration.

22.8.1. CRUDRepository

`CRUDRepository` delegates to the configured `TypeRepresentationStrategy` (see Entity type representation) for type based queries.

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

`EndResult<T> findAll()` `EndResult<T> findAll(Sort)` and `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 `deleteAll()`

22.8.2. IndexRepository and NamedIndexRepository

`IndexRepository` works with the indexing subsystem and provides methods to find entities by indexed properties, ranged queries, and combinations thereof. The index key is the name of the indexed entity field, unless overridden in the `@Indexed` annotation.

Iterate over all indexed entity instances with a certain field value

```
EndResult<T> findAllByPropertyValue(key, value)
```

Get a single entity instance with a certain field value

```
T findByPropertyValue(key, value)
```

Iterate over all indexed entity instances with field values in a certain numerical range (inclusive)

```
EndResult<T> findAllByRange(key, from, to)
```

Iterate over all indexed entity instances with field values matching the given fulltext string or QueryContext query

```
EndResult<T> findAllByQuery(key, queryOrQueryContext)
```

There is also a `NamedIndexRepository` with the same methods, but with an additional index name parameter, making it possible to query any index.

22.8.3. TraversalRepository

`TraversalRepository` delegates to the Neo4j traversal framework.

Iterate over a traversal result

```
Iterable<T> findAllByTraversal(startEntity, traversalDescription)
```

22.8.4. Query and Finder Methods

Annotated queries

Queries using the Cypher graph query language can be supplied with the `@Query` annotation. That means every method annotated with `@Query("start n=node:IndexName(key={node or 0}) match (n)-(m) return m")` will use the supplied query string. The named or indexed parameter `{node}` will be substituted by the actual method parameter. Node and Relationship-Entities are handled directly, Iterables thereof as well. All other parameters are replaced directly (i.e. Strings, Longs, etc). There is special support for the `Sort` and `Pageable` parameters from Spring Data Commons, which are supported to add programmatic paging, slicing and sorting (alternatively static paging and sorting can be supplied in the query string itself). For using the named parameters you have to either annotate the parameters of the method with the `@Param("node")` annotation or enable debug symbols. Indexed parameters are always usable.

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 `totalCount` property of the returned `Page`.

Named queries

Spring Data Neo4j also supports the notion of named queries which are externalized in property-config-files (`META-INF/neo4j-named-queries.properties`). Those files have the format: `Entity.finderName=query` (e.g. `Person.findBoss=start p=node({0}) match (p) [:BOSS]-(boss) return boss`). Otherwise named queries support the same parameters as annotated queries. For count queries the lookup name is `Entity.finderName.count=count-query`. The default query lookup names can be overriden by using an `@Query` annotation with `queryName="my-query-name"` or `countQueryName="my-query-name"`.

Query results

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

Cypher examples

There is a screencast available showing many features of the query language. The following examples are taken from the cineasts dataset of the tutorial section.

```
start n=node(0) return n
```

returns the node with id 0

```
start movie=node:Movie(title='Matrix') return movie
```

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

```
start movie=node:Movie(title='Matrix') match (movie)  [:ACTS_IN]-(actor) return actor.name
```

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

```
start movie=node:Movie(title='Matrix') match (movie)  [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'

```
start user=node:User(login='micha') match (user)-[: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 'micha', aggregated by movie.title, with averaged ratings and rating-counts sorted by both

Example 71. Examples of Cypher queries placed on repository methods with @Query where values are replaced with method parameters, as described in the Annotated queries) section.

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

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

    // returns the nodes which will use index named title equal to movieTitle
parameter
    // movieTitle String must not contain any spaces, otherwise you will receive a
NullPointerException.
    @Query("start movie=node:Movie(title={0}) return movie")
    Movie getMovieFromTitle(String movieTitle);

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

    // returns users who rated a movie (movie parameter) higher than rating (rating
parameter)
    @Query("start movie=node:({0}) " +
           "match (movie)<-[r:RATED]-(user) " +
           "where r.stars > {1} " +
           "return user")
    Iterable<User> getUsersWhoRatedMovieFromTitle(Movie movie, Integer rating);

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

Queries derived from finder-method names

As known from Rails or Grails it is possible to derive queries for domain entities from finder method names like `Iterable<Person> findByNameAndAgeGreaterThan(String name, int age)`. Using the

infrastructure in Spring Data Commons that allows to collect the meta information about entities and their properties a finder method name can be split into its semantic parts and converted into a cypher query. `@Indexed` fields will be converted into index-lookups of the `start` clause, navigation along relationships will be reflected in the `match` clause properties with operators will end up as expressions in the `where` clause. Order and limiting of the query will by handled by provided `Pageable` or `Sort` parameters. The other 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.

Example 72. Some examples of methods and resulting Cypher queries of a PersonRepository

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

    // start person=node:Person(id={0}) return person
    Person findById(String id)

    // start person=node:Person({0}) return person - {0} will be "id:"+name
    Iterable<Person> findByNameLike(String name)

    // start person=node:__types__("className"="com...Person")
    // where person.age = {0} and person.married = {1}
    // return person
    Iterable<Person> findByAgeAndMarried(int age, boolean married)

    // start person=node:__types__("className"="com...Person")
    // match person<-[:CHILD]-parent
    // where parent.age > {0} and person.married = {1}
    // return person
    Iterable<Person> findByParentAgeAndMarried(int age, boolean married)
}
```

Derived Finder Methods

Use the meta information of your domain model classes to declare repository finders that navigate along relationships and compare properties. The path defined with the method name is used to create a Cypher query that is executed on the graph.

Example 73. Repository and usage of derived finder methods

```
@NodeEntity
public static class Person {
    @GraphId Long id;
    private String name;
    private Group group;

    private Person(){}
    public Person(String name) {
        this.name = name;
    }
}

@NodeEntity
public static class Group {
    @GraphId Long id;
    private String title;
    // incoming relationship for the person -> group
    @RelatedTo(type = "group", direction = Direction.INCOMING)
    private Set<Person> members=new HashSet<Person>();

    private Group(){}
    public Group(String title, Person...people) {
        this.title = title;
        members.addAll(asList(people));
    }
}
public interface PersonRepository extends GraphRepository<Person> {
    Iterable<Person> findByGroupTitle(String name);
}

@Autowired PersonRepository personRepository;

Person oliver=personRepository.save(new Person("Oliver"));
final Group springData = new Group("spring-data",oliver);
groupRepository.save(springData);

final Iterable<Person> members = personRepository.findByGroupTitle("spring-data");
assertThat(members.iterator().next().name, is(oliver.name));
```

22.8.5. Cypher-DSL repository

Spring Data Neo4j supports the new Cypher-DSL to write Cypher queries in a statically typed way. Just by including `CypherDslRepository` to your repository you get the `Page<T> query(Execute query, params,`

`Pageable page)`, `Page<T> query(Execute query, Execute countQuery, params, Pageable page)` and the `EndResult<T> query(Execute query, params);`. The result type of the Cypher-DSL builder is called `Execute`.

Example 74. Examples for Cypher-DSL repository

```
import static org.neo4j.cypherdsl.CypherQuery.*;
import static org.neo4j.cypherdsl.querydsl.CypherQueryDSL.*;

public interface PersonRepository extends GraphRepository<Person>,
  CypherDslRepository<Person> {}

@Autowired PersonRepository repo;
// START company=node:Company(name={name}) MATCH company<-[:WORKS_AT]->person RETURN
person

Execute query = start( lookup( "company", "Company", "name", param("name") ) ).
                        match( path().from( "company" ).in( "WORKS_AT" ).to(
"person" )).
                        returns( identifier( "person" ))
Page<Person> people = repo.query(query , map("name","Neo4j"), new PageRequest(1,10));

QPerson person = QPerson.person;
QCompany company = QCompany.company;
Execute query = start( lookup( company, "Company", company.name, param("name") ) ).
                        match( path().from( company ).in( "WORKS_AT" ).to( person
).
                        .where(person.firstName.like("P*").and(person.age.gt(25))).
                        returns( identifier(person) )
EndResult<Person> people = repo.query(query , map("name","Neo4j"));
```

22.8.6. Cypher-DSL and QueryDSL

To use Cypher-DSL with Query-DSL the Mysema dependencies have to be declared explicitly as they are optional in the Cypher-DSL project.

```
<dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-core</artifactId>
    <version>2.2.3</version>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-lucene</artifactId>
    <version>2.2.3</version>
    <optional>true</optional>
    <exclusions>
        <exclusion>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
    <version>2.2.3</version>
    <scope>provided</scope>
</dependency>
```

It is possible to use the Cypher-DSL along with the predicates and code generation features of the QueryDSL project. This will allow you to use Java objects as part of the query, rather than strings, for the names of properties and such. In order to get this to work you first have to add a code processor to your Maven build, which will parse your domain entities marked with @NodeEntity, and from that generate QPerson-style classes, as shown in the previous section. Here is what you need to include in your Maven POM file.

```
<plugin>
    <groupId>com.mysema.maven</groupId>
    <artifactId>maven-apt-plugin</artifactId>
    <version>1.0.2</version>
    <configuration>
        <processor>org.springframework.data.neo4j.querydsl.SDNAnnotationProcessor</proces
sor>
    </configuration>
    <executions>
        <execution>
            <id>test-sources</id>
            <phase>generate-test-sources</phase>
            <goals>
                <goal>test-process</goal>
            </goals>
            <configuration>
                <outputDirectory>target/generated-sources/test</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This custom QueryDSL AnnotationProcessor will generate the query classes that can be used when constructing Cypher-DSL queries, as in the previous section.

22.8.7. Creating repositories

The `Repository` instances should normally be injected but can also be created manually via the `Neo4jTemplate`.

Example 75. Using basic GraphRepository methods

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

@Autowired PersonRepository repo;
// OR
GraphRepository<Person> repo = template
                            .repositoryFor(Person.class);

Person michael = repo.save(new Person("Michael", 36));

Person dave = repo.findOne(123);

Long numberOfPeople = repo.count();

EndResult<Person> devs = graphRepository.findAllByPropertyValue("occupation",
"developer");

EndResult<Person> middleAgedPeople = graphRepository.findAllByRange("age", 20, 40);

EndResult<Person> aTeam = graphRepository.findAllByQuery("name", "A*");

Iterable<Person> aTeam = repo.findAllByQuery("name", "A*");

Iterable<Person> davesFriends = repo.findAllByTraversal(dave,
    Traversal.description().pruneAfterDepth(1)
    .relationships(KNOWS).filter(returnAllButStartNode()));
```

22.8.8. Composing repositories

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

Example 76. Composing repositories

```
public interface PersonRepository extends GraphRepository<Person>,
PersonRepositoryExtension {}

// configure the repositories, preferably via the neo4j:repositories namespace
// (template reference is optional)
<neo4j:repositories base-package="org.example.repository"
    graph-database-context-ref="template"/>

// have it injected
@Autowired
PersonRepository personRepository;
// or created via the template
PersonRepository personRepository = template.repositoryFor(Person.class);


Person michael = personRepository.save(new Person("Michael",36));

Person dave=personRepository.findOne(123);

Iterable<Person> devs = personRepository.findAllByPropertyValue("occupation",
"developer");

Iterable<Person> aTeam = graphRepository.findAllByQuery( "name","A*");

Iterable<Person> friends = personRepository.findFriends(dave);


// alternatively select some of the required repositories individually
public interface PersonRepository extends CRUDGraphRepository<Node,Person>,
        IndexQueryExecutor<Node,Person>, TraversalQueryExecutor<Node,Person>,
        PersonRepositoryExtension {}

// provide a custom extension if needed
public interface PersonRepositoryExtension {
    Iterable<Person> findFriends(Person person);
}

public class PersonRepositoryImpl implements PersonRepositoryExtension {
    // optionally inject default repository, or use DirectGraphRepositoryFactory
    @Autowired PersonRepository baseRepository;
    public Iterable<Person> findFriends(Person person) {
        return baseRepository.findAllByTraversal(person, friendsTraversal);
    }
}

// configure the repositories, preferably via the datagraph:repositories namespace
// (template reference is optional)
```

```
<neo4j:repositories base-package="org.springframework.data.neo4j"
    graph-database-context-ref="template"/>

// have it injected
@Autowired
PersonRepository personRepository;

Person michael = personRepository.save(new Person("Michael",36));

Person dave=personRepository.findOne(123);

EndResult<Person> devs = personRepository.findAllByPropertyValue("occupation",
"developer");

EndResult<Person> aTeam = graphRepository.findAllByQuery( "name","A*");

Iterable<Person> friends = personRepository.findFriends(dave);
```

NOTE If you use `<context:component-scan>` in your spring config, please make sure to put it behind `<neo4j:repositories>`, as the RepositoryFactoryBean adds new bean definitions for all the declared repositories, the context scan doesn't pick them up otherwise.

22.9. Conversion

`Neo4jTemplate` has a generic convert method which might also use projection underneath. The same conversion facilities that are used by default in the result handling DSL are offered here for individual use.

Supported conversions are: Nodes to Paths and to Entities, Relationships to Paths and to Entities, Paths to Node (EndNode), Relationships (LastRelationship) and to EntityPaths. Entities to Nodes or Relationships.

It is also possible to provide a custom ResultConverter that additionally takes care of conversions.

22.9.1. Mapping Query Results

For both queries executed via the result conversion DSL as well as repository methods, it is possible to specify a conversion of complex query results to POJO interfaces or objects. Those result objects are then populated with the query result data and can be serialized and sent to a different part of the applicaton, e.g. a frontend-ui.

Use an interface annotated with `@QueryResult` and getter methods which might be annotated with `@ResultColumn("columnName")` to match the query-result column names. Or use a plain POJO, both work.

Example 77. Example of query result mapping

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

    @Query("START movie=node:Movie(id={0})
            MATCH movie-[rating?:rating]->(),
                  movie<-[:ACTS_IN]-actor
            RETURN movie, COLLECT(actor), AVG(rating.stars)")
    MovieData getMovieData(String movieId);

    @QueryResult
    public class MovieData {
        Movie movie;

        @ResultColumn("AVG(rating.stars)")
        Double rating;

        @ResultColumn("COLLECT(actor)")
        Collection<Actor> cast;
    }

    // alternatively use
    @QueryResult
    public interface MovieData {
        @ResultColumn("movie")
        Movie getMovie();

        @ResultColumn("AVG(rating.stars)")
        Double getRating();

        @ResultColumn("COLLECT(actor)")
        Iterable<Actor> getCast();
    }
}
```

22.10. Projecting entities

As the underlying data model of a graph database doesn't imply and enforce strict type constraints like a relational model does, it offers much more flexibility on how to model your domain classes and which of those to use in different contexts.

For instance an order can be used in these contexts: customer, procurement, logistics, billing, fulfillment and many more. Each of those contexts requires its distinct set of attributes and operations. As Java doesn't support mixins one would put the sum of all of those into the entity class and thereby

making it very big, brittle and hard to understand. Being able to take a basic order and project it to a different (not related in the inheritance hierarchy or even an interface) order type that is valid in the current context and only offers the attributes and methods needed here would be very beneficial.

Spring Data Neo4j offers initial support for projecting node and relationship entities to different target types. All instances of this projected entity share the same backing node or relationship, so changes are reflected on the same data.

This could for instance also be used to handle nodes of a traversal with a unified (simpler) type (e.g. for reporting or auditing) and only project them to a concrete, more functional target type when the business logic requires it.

Example 78. Projection of entities

```
@NodeEntity
class Trainee {
    String name;
    @RelatedTo
    Set<Training> trainings;
}

for (Person person : graphRepository.findAllByPropertyValue("occupation","developer"
)) {
    Developer developer = person.projectTo(Developer.class);
    if (developer.isJavaDeveloper()) {
        trainInSpringData(developer.projectTo(Trainee.class));
    }
}
```

22.11. Geospatial Queries

`SpatialRepository` is a dedicated Repository for spatial queries. Spring Data Neo4j provides an optional dependency to neo4j-spatial which is an advanced library for GIS operations. So if you include the maven dependency in your `pom.xml`, Neo4j-Spatial and the required `SPATIAL` index provider is available.

Spring Data Neo4j integrates with the common geo-spatial types in Spring Data Commons, like `Circle, Box, Polygon and Point`. You can use them as parameters for repository finder methods and property values. We also provide converters between WKT and `Shape`/`Point` objects.

Example 79. Neo4j-Spatial Dependencies

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-spatial</artifactId>
    <version>0.13-neo4j-2.0.1</version>
</dependency>
```

To have your entities available for spatial index queries, please include a String property containing a "well known text", location string. WKT is the Well Known Text Spatial Format eg. `POINT( LON LAT )` or `POLYGON`

Alternatively you can also use one of the geo-spatial primitives from Spring Data Commons which are automatically converted to WKT when stored in the graph.

Example 80. Fields of Well Known Text

```
@NodeEntity
class Venue {
   String name;
   @Indexed(type = POINT, indexName = "VenueLocation") String wkt;
   public void setLocation(float lon, float lat) {
      this.wkt = String.format("POINT( %.2f %.2f )",lon,lat);
   }
}

venue.setLocation(15,56);
```

Example 81. Field of Point

```
@NodeEntity
class Venue {
   String name;
   @Indexed(type = POINT, indexName = "VenueLocation") Point wkt;
}

venue.setLocation(new Point(15,56));
```

After adding the `SpatialRepository` to your repository you can use the `findWithinBoundingBox`, `findWithinDistance`, `findWithinWellKnownText`, `findWithinShape` methods. We recommend to use the methods using the geospatial primitives as they are more typesafe and less error-prone in mixing the

coordinate order.

Example 82. Spatial Queries

```
Iterable<Person> teamMembers = personRepository.findWithinBoundingBox("personLayer",
55, 15, 57, 17);
Iterable<Person> teamMembers = personRepository.findWithinBoundingBox("personLayer",
new Box(new Point(15,55), new Point(17,57));
Iterable<Person> teamMembers = personRepository.findWithinWellKnownText("personLayer
", "POLYGON ((15 55, 15 57, 17 57, 17 55, 15 55))");
Iterable<Person> teamMembers = personRepository.findWithinShape("personLayer", new
Polygon(new Point(15,55),new Point(15,57), new Point(17,57),new Point(17,55)));
Iterable<Person> teamMembers = personRepository.findWithinDistance("personLayer", 16
,56,70);
Iterable<Person> teamMembers = personRepository.findWithinDistance("personLayer", new
Circle(new Point(16,56),new Distance(70, Metrics.KILOMETERS)));
```

Example 83. Methods of the Spatial Repository

```
public interface SpatialRepository<T> {
    @Transactional
    EndResult<T> findWithinBoundingBox(String indexName, double lowerLeftLat,
                                       double lowerLeftLon,
                                       double upperRightLat,
                                       double upperRightLon);

    @Transactional
    EndResult<T> findWithinBoundingBox(String indexName, Box box);

    @Transactional
    EndResult<T> findWithinDistance( final String indexName, final double lat, double
lon, double distanceKm);

    @Transactional
    EndResult<T> findWithinDistance( final String indexName, Circle circle);

    @Transactional
    EndResult<T> findWithinWellKnownText( final String indexName, String
wellKnownText);

    /**
     * Converts the shape into a well-known text representation and executes the
appropriate WKT query
     */
    @Transactional
    EndResult<T> findWithinShape( final String indexName, Shape shape);
}
```

Example 84. Derived Spatial Finder Methods

```
public interface PersonRepository extends GraphRepository<Person>, SpatialRepository
<Person> {
    Collection<Person> findByWktNearAndName(Circle circle, String name);
    Collection<Person> findByWktWithinAndAgeGreaterThan(Circle circle, int age);
    Collection<Person> findByWktWithinAndPersonality(Polygon polygon, Personality
personality);
    Collection<Person> findByWktWithin(Box box);
}
```

22.12. Active Record Methods for Advanced Mapping Mode

This chapter only applies to the advanced mapping. Currently the Aspects introduce the following methods by default, this will change in the future, there will be separate Mixin-Interfaces that can selectively be mixed into the domain entities if needed. Otherwise the AspectJ interaction will be restricted to field access interception and post-constructor handling.

The node and relationship aspects introduce (via AspectJ ITD - inter-type declaration) several methods to the entities.

Persisting the node entity after creation and after changes outside of a transaction. Participates in an open transaction, or creates its own implicit transaction otherwise.

`nodeEntity.persist()`

Accessing node and relationship IDs

`nodeEntity.getNodeId()` and `relationshipEntity.getRelationshipId()`

Accessing the node or relationship backing the entity

`entity.getPersistentState()`

equals() and hashCode() are delegated to the underlying state

`entity.equals()` and `entity.hashCode()`

Creating relationships to a target node entity, and returning the relationship entity instance

`nodeEntity.relateTo(targetEntity, relationshipClass, relationshipType)`

Retrieving a single relationship entity

`nodeEntity.getRelationshipTo(targetEntity, relationshipClass, relationshipType)`

Creating relationships to a target node entity and returning the relationship

`nodeEntity.relateTo(targetEntity, relationshipType)`

Retrieving a single relationship

`nodeEntity.getRelationshipTo(targetEnttiy, relationshipType)`

Removing a single relationship

`nodeEntity.removeRelationshipTo(targetEntity, relationshipType)`

Remove the node entity, its relationships, and all index entries for it

`nodeEntity.remove()` and `relationshipEntity.remove()`

Project entity to a different target type, using the same backing state

`entity.projectTo(targetClass)`

Traverse, starting from the current node. Returns end nodes of traversal converted to the provided type.

```
nodeEntity.findAllByTraversal(targetType, traversalDescription)
```

Traverse, starting from the current node. Returns `EntityPath`s of the traversal result bound to the provided start and end-node-entity types

```
Iterable<EntityPath> findAllPathsByTraversal(traversalDescription)
```

Executes the given Cypher query, providing the `{self}` variable with the node-id and returning the results converted to the target type.

```
<T> Iterable<T> NodeBacked.findAllByQuery(final String query, final Class<T> targetType)
```

Executes the given query, providing `{self}` variable with the node-id and returning the original result, but with nodes and relationships replaced by their appropriate entities.

```
Iterable<Map<String,Object>> NodeBacked.findAllByQuery(final String query)
```

Executes the given query, providing `{self}` variable with the node-id and returns a single result converted to the target type.

```
<T> T NodeBacked.findByQuery(final String query, final Class<T> targetType)
```

22.13. Transactions

Neo4j is a transactional database, only allowing modifications to be performed within transaction boundaries. Reading data does however not require transactions. Spring Data Neo4j integrates nicely with both the declarative transaction support with `@Transactional` as well as the manual transaction handling with `TransactionTemplate`. It also supports the rollback mechanisms of the Spring Testing library.

Spring Data Neo4j integrates with transaction managers configured using Spring. The simplest scenario of just running the graph database uses a `SpringTransactionManager` provided by the Neo4j kernel to be used with Spring's `JtaTransactionManager`. That is, configuring Spring to use Neo4j's transaction manager.

NOTE To avoid name collisons the transaction manager configured by Spring Data Neo4j is called `neo4jTransactionManager` and is aliased to `transactionManager`. So defining a separate `transactionManager` bean should not interfere with Spring Data Neo4j operations.

NOTE The explicit XML configuration given below is encoded in the `Neo4jConfiguration` configuration bean that uses Spring's `@Configuration` feature. This greatly simplifies the configuration of Spring Data Neo4j.

Example 85. Simple transaction manager configuration

```
<bean id="neo4jTransactionManager"
        class="org.springframework.data.neo4j.config.JtaTransactionManagerFactoryBean
">
    <constructor-arg ref="graphDatabaseService"/>
</bean>

<tx:annotation-driven mode="aspectj" transaction-manager="neo4jTransactionManager"/>
```

For scenarios with multiple transactional resources there are two options. The first option is to have Neo4j participate in the externally configured transaction manager using the Spring support in Neo4j by enabling the configuration parameter for your graph database. Neo4j will then use Spring's transaction manager instead of its own.

Example 86. Neo4j Spring integration

```
<context:annotation-config />
<context:spring-configured/>

<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager">
        <bean id="jotm" class=
"org.springframework.data.neo4j.transaction.JotmFactoryBean"/>
    </property>
</bean>

<bean id="graphDatabaseService" class="org.neo4j.kernel.EmbeddedGraphDatabase"
destroy-method="shutdown">
    <constructor-arg value="target/test-db"/>
    <constructor-arg>
        <map>
            <entry key="tx_manager_impl" value="spring-jta"/>
        </map>
    </constructor-arg>
</bean>

<tx:annotation-driven mode="aspectj" transaction-manager="transactionManager"/>
```

One can also configure a stock XA transaction manager (e.g. Atomikos, JOTM, App-Server-TM) to be used with Neo4j and the other resources. For a bit less secure but fast 1-phase-commit-best-effort, use `ChainedTransactionManager`, which comes bundled with Spring Data Neo4j. It takes a list of transaction

managers as constructor params and will handle them in order for transaction start and commit (or rollback) in the reverse order.

Example 87. ChainedTransactionManager example

```
<bean id="jpaTransactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<bean id="jtaTransactionManager"
        class="org.springframework.data.neo4j.config.JtaTransactionManagerFactoryBean
">
    <constructor-arg ref="graphDatabaseService"/>
</bean>
<bean id="transactionManager"
        class="org.springframework.data.neo4j.transaction.ChainedTransactionManager">
    <constructor-arg>
        <list>
            <ref bean="jpaTransactionManager"/>
            <ref bean="jtaTransactionManager"/>
        </list>
    </constructor-arg>
</bean>

<tx:annotation-driven mode="aspectj" transaction-manager="transactionManager"/>
```

22.14. Detached node entities in advanced mapping mode

This section only applies to the advanced mapping (AspectJ-backed). The simple mapping always detaches entities on load as it copies the data out of the graph into the entities and stores it back fully too.

Node entities can be in two different persistence states: attached or detached. By default, newly created node entities are in the detached state. When `persist() or template.save()` is called on the entity, it becomes attached to the graph, and its properties and relationships are stores in the database. If the save operation is not called within a transaction, it automatically creates an implicit transaction only for the operation.

Changing an attached entity inside a transaction will immediately write through the changes to the datastore. Whenever an entity is changed outside of a transaction it becomes detached. The changes are stored in the entity (its fields) itself until the next call to a save operation.

All entities returned by library functions are initially in an attached state. Just as with any other entity,

changing them outside of a transaction detaches them, and they must be reattached with `persist()` for the data to be saved.

Example 88. Persisting entities

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

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

22.14.1. Relating detached entities

As mentioned above, an entity simply created with the `new` keyword starts out detached. It also has no state assigned to it. If you create a new entity with `new` and then throw it away, the database won't be touched at all.

Now consider this scenario:

Example 89. Relationships outside of transactions

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

@NodeEntity
class Actor {
}

Movie movie = new Movie();
Actor actor = new Actor();

movie.setTopActor(actor);
```

Neither the actor nor the movie has been assigned a node in the graph. If we were to call `movie.persist()`, 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 call actor.persist() 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.

Important to note here is that if `actor.persist()` 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 behavior 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.

The save operation (merge) stores all properties of the entity to the graph database and puts the entity in attached mode. There is no need to update the reference to the Java POJO as the underlying backing node handles the read-through transparently. If multiple object instances that point to the same node are persisted, the ordering is not important as long as they contain distinct changes. For concurrent changes a concurrent modification exception is thrown (subject to be parameterized in the future).

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 of `persist()` is however only cascaded 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 90. Cascade for modified fields

```
actor.setName("Billy Bob");
movie.persist();
```

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

22.15. Entity type representation

There are several ways to represent the Java type hierarchy of the data model in the graph. In general, for all node and relationship entities, type information is needed to perform certain repository operations (such as findAll()). This means that some of the type information hierarchy needs to be saved in the graph database, and SDN makes use of "Type Representation Strategies" in order to achieve this.

Implementations of `TypeRepresentationStrategy` take care of persisting this information during entity instance creation. They are also used by certain repository methods to perform their operations, like `findAll` and `count`. The derived finderMethods also use the type information for graph global queries.

There are four available implementations for node entities to choose from, Spring Data Neo4j defaults to the label based strategy.

- `LabelBasedNodeTypeRepresentationStrategy` this is the default strategy used.

  Stores entity types in node labels. Each node gets labeled with its type and all supertypes and interfaces that are also `@NodeEntity`-annotated. There is a special Label prefixed with _ that represents the current type of the entity.

- `IndexingNodeTypeRepresentationStrategy`

  Stores entity types in the integrated index. Each entity node gets indexed with its type and all supertypes and interfaces that are also `@NodeEntity`-annotated. The special index used for this is named `types`. Additionally, in order to retrieve the type of an entity node, each node has a property `type` with the fully qualified type of that entity.

- `SubReferenceNodeTypeRepresentationStrategy`

  Stores entity types in a tree in the graph representing the type and interface hierarchy. Each entity has a INSTANCE_OF relationship to a type node representing that entity's type. The type may or may not have a SUBCLASS_OF relationship to another type node.

- `NoopNodeTypeRepresentationStrategy`

  Does not store any type information, and does hence not support finding by type, counting by type, or retrieving the type of any entity.

There are two implementations for relationship entities available, with the same behavior as the corresponding ones above:

- `IndexingRelationshipTypeRepresentationStrategy`

  Stores relationship entity types in the integrated index. Each entity relationship gets indexed with its type and all supertypes and interfaces that are also `@RelationshipEntity`-annotated. The special index used for this is named `rel_types`. Additionally, in order to retrieve the type of an entity relationship, each relationship has a property `type` with the fully qualified type of that entity.

- `NoopRelationshipTypeRepresentationStrategy`

  In order to use a different Type Representation Strategy, simply register an alternative "typeRepresentationStrategyFactory" spring bean specifying the strategy required. For example to use the legacy indexing strategy for nodes you could define the following override bean.

Example 91. XML-based configuration

```
<bean id="typeRepresentationStrategyFactory" class=
"org.springframework.data.neo4j.support.typerepresentation.TypeRepresentationStrategy
Factory">
    <constructor-arg ref="graphDatabase"/>
    <constructor-arg value="Indexed"/>
</bean>
```

Whilst in Java Config this may look as follows:

Example 92. Java-based configuration

```
@Configuration
@EnableNeo4jRepositories(basePackages = "org.example.repositories")
static class Config extends Neo4jConfiguration {

    Config() {
        // Equivalent of setting basePackage for XML based <neo4j:config base-
package=".."/>
        // (This will probably move into an/the @EnableNeo4jRepositories in the
future)
        setBasePackage("org.example.domain");
    }

    @Override
    public TypeRepresentationStrategyFactory typeRepresentationStrategyFactory() {
        return new TypeRepresentationStrategyFactory(
                    graphDatabase(),
                    TypeRepresentationStrategyFactory.Strategy.Indexed);
    }

    ...

}
```

As some type information is also stored in labels, node/relationship-properties and/or indexes it might amount to a substantial amount of data in the graph. It is possible to use an `@TypeAlias("name")` annotation on nodes and relationships to have a short constant name for each type which is (unlike the default approach) renaming-refactoring-safe. From 3.0 onwards, Spring Data Neo4j uses the simple class name as the default whilst previous versions used to default to the fully qualified name. If you would like to use the fully qualified class name by default, you can

- Register a `Neo4jMappingContext` bean configured with an instance of `org.springframework.data.neo4j.support.mapping.ClassNameAlias`
- Override the spring "entityAlias" bean with an instance of `org.springframework.data.neo4j.support.mapping.ClassNameAlias`. For example, using XML config this would look as follows:

```
<bean id="entityAlias" class=
"org.springframework.data.neo4j.support.mapping.ClassNameAlias" />
```

It is also possible to opt out of storing type information completely by using the `NoopTypeRepresentationStrategies`.

Spring Data Neo4j will by default autodetect which are the most suitable strategies for node and relationship entities. For new data stores, it will always opt for the indexing strategies (Label based for nodes, and legacy index based for relationships). If a data store was created with the older`SubReferenceNodeTypeRepresentationStrategy`, then it will continue to use that strategy for node entities. It will however in that case use the no-op strategy for relationship entities, which means that the old data stores have no support for searching for relationship entities. The indexing strategies are recommended for all new users.

22.15.1. Entity type safety

While some methods such as findAll and count will used the stored type information to return uniformly typed entities, some others such as findOne will fetch the requested node regardless of its type. This may result in odd behaviors where a repository extending GraphRepository<T> may return entities of type other than T (see example below).

Example 93. Requests to repository without type safety enforcement

```
@NodeEntity
public class Dog {
    @GraphId Long nodeId
}

@NodeEntity
public class Cat {
    @GraphId Long nodeId
}

public interface CatRepository extends GraphRepository<Cat> {}

public interface DogRepository extends GraphRepository<Dog> {}

catRepository.save(new Cat());
dogRepository.save(new Dog());

GET /dogs/0 // returns a Cat
GET /dogs/1 // returns a Dog
```

In order to customize the type safety enforcement applied when creating entities from the requested nodes in the graph, you might want to register a different "typeSafetyPolicy" spring bean specifying the TypeSafetyOption to use.

There are 3 different type safety options available :

- TypeSafetyOption.NONE : Sets the system to not be type safe.
- TypeSafetyOption.RETURNS_NULL : Sets the system to return null if a entity should be loaded which is not of the requested type.
- TypeSafetyOption.THROWS_EXCEPTION : Sets the system to throw an exception if a entity should be loaded which is not of the requested type (default setting).

The TypeSafetyPolicy override bean can be declared in the following way :

Example 94. TypeSafetyPolicy XML-based configuration

```
<bean id="typeSafetyPolicy" class=
"org.springframework.data.neo4j.support.typesafety.TypeSafetyPolicy">
    <constructor-arg value="RETURNS_NULL" />
</bean>
```

Or in java config :

Example 95. TypeSafetyPolicy Java-based configuration

```
@Configuration
@EnableNeo4jRepositories(basePackages = "org.example.repositories")
static class Config extends Neo4jConfiguration {

    Config() {
        // Equivalent of setting basePackage for XML based <neo4j:config base-
package=".."/>
        // (This will probably move into an/the @EnableNeo4jRepositories in the
future)
        setBasePackage("org.example.domain");
    }

    @Override
    public TypeSafetyPolicy typeSafetyPolicy() throws Exception {
        return new TypeSafetyPolicy(TypeSafetyOption.RETURNS_NULL);
    }

    ...

}
```

22.16. Bean validation (JSR-303)

Spring Data Neo4j supports property-based validation support. When a property is changed and persisted, it is checked against the annotated constraints, e.g. `@Min`, `@Max`, `@Size`, etc. Validation errors throw a `ValidationException`. The validation support that comes with Spring is used for evaluating the constraints. To use this feature, a validator has to be registered with the `Neo4jTemplate`, which is done automatically by the `Neo4jConfiguration` if one is present in the Spring Config.

Example 96. Bean validation

```
@NodeEntity
class Person {
    @Size(min = 3, max = 20)
    String name;

    @Min(0) @Max(100)
    int age;
}
```

The validation supports needs the bean validation API and a reference implementation configured. Right now this is the Hibernate Validator by default (which is not integrated with Hibernate ORM). The maven dependency is:

Example 97. Validation setup

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.2.0.Final</version>
</dependency>

// the application-context should contain a LocalValidatorFactoryBean

<bean id="validator"
  class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

Chapter 23. Environment setup

Spring Data Neo4j dramatically simplifies development, but some setup is naturally required. For building the application, Maven needs to be configured to include the Spring Data Neo4j dependencies. For the advanced mapping mode, it is necessary to configure the AspectJ weaving. After the build setup is complete, the Spring application needs to be configured to make use of Spring Data Neo4j. Examples for these different setups can be found in the Spring Data Neo4j examples.

Spring Data Neo4j projects can be built using Maven. There are also means to build them with Gradle or Ant/Ivy.

23.1. Dependencies for Spring Data Neo4j Simple Mapping

For the simple POJO mapping it is enough to add the `org.springframework.data:spring-data-neo4j:3.3.0.M1` dependency to your project.

Example 98. Maven dependencies for Spring Data Neo4j

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-neo4j</artifactId>
<version>3.3.0.M1</version>
</dependency>
```

23.2. Gradle configuration for Advanced Mapping (AspectJ)

The necessary build plugin to build Spring Data Neo4j projects with Gradle is available as part of the Spring Data Neo4j distribution or on Github which makes the usage as easy as:

Example 99. Gradle Build Configuration

```
sourceCompatibility = 1.6
targetCompatibility = 1.6

springVersion = "4.0.7.RELEASE"
springDataNeo4jVersion = "3.3.0.M1"
aspectjVersion = "1.7.4"

apply from:'https://github.com/SpringSource/spring-data-neo4j/raw/master/build/
gradle/springdataneo4j.gradle'

configurations {
    runtime
    testCompile
}
repositories {
    mavenCentral()
mavenLocal()
mavenRepo urls: "http://maven.springframework.org/release"
}
```

The actual `springdataneo4j.gradle` is very simple, just decorating the `javac` tasks with the `iajc` ant task.

23.3. Ant/Ivy configuration for Advanced Mapping (AspectJ)

The supplied sample ant build configuration is mainly about resolving the dependencies for Spring Data Neo4j Aspects and AspectJ using Ivy and integrating the iajc ant task in the build.

Example 100. Ant/Ivy Build Configuration

```
<taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties"
classpath="${lib.dir}/aspectjtools.jar"/>

<target name="compile" description="Compile production classes" depends="
lib.retrieve">
<mkdir dir="${main.target}" />

<iajc sourceroots="${main.src}" destDir="${main.target}" classpathref="path.libs"
source="1.6">
<aspectpath>
<pathelement location="${lib.dir}/spring-aspects.jar"/>
</aspectpath>
<aspectpath>
<pathelement location="${lib.dir}/spring-data-neo4j-aspects.jar"/>
</aspectpath>
</iajc>
</target>
```

23.4. Maven configuration for Advanced Mapping

Spring Data Neo4j projects are easiest to build with Apache Maven. The core dependency is Spring Data Neo4j Aspects which comes with transitive dependencies to Spring Data Neo4j, Spring Data Commons, parts of the Spring Framework, AspectJ and the Neo4j graph database.

23.4.1. Repositories

The milestone releases of Spring Data Neo4j are available from the dedicated milestone repository. Neo4j releases and milestones are available from Maven Central.

Example 101. Spring milestone repository

```
<repository>
<id>spring-libs-milestone</id>
<name>Spring Milestone</name>
<url>http://repo.spring.io/libs-milestone</url>
</repository>
```

23.4.2. Dependencies

The dependency on `spring-data-neo4j-aspects` will transitively pull in the necessary parts of Spring

Framework (core, context, aop, aspects, tx), AspectJ, Neo4j, and Spring Data Commons. If you already use these (or different versions of these) in your project, then include those dependencies on your own. In this case, please make sure that the versions match.

Example 102. Maven dependencies

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j-aspects</artifactId>
    <version>3.3.0.M1</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.8.2</version>
</dependency>
```

23.4.3. Maven AspectJ build configuration

Since the advanced mapping uses AspectJ for build-time aspect weaving of entities, it is necessary to hook the AspectJ Maven plugin into the build process. The plugin also has its own dependencies. You also need to explicitly specify the aspect libraries (spring-aspects and spring-data-neo4j-aspects).

Example 103. AspectJ configuration

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.4</version>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.8.2</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>1.8.2</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>test-compile</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <outxml>true</outxml>
        <aspectLibraries>
            <aspectLibrary>
                <groupId>org.springframework</groupId>
                <artifactId>spring-aspects</artifactId>
            </aspectLibrary>
            <aspectLibrary>
                <groupId>org.springframework.data</groupId>
                <artifactId>spring-data-neo4j-aspects</artifactId>
            </aspectLibrary>
        </aspectLibraries>
        <source>1.7</source>
        <target>1.7</target>
    </configuration>
</plugin>
```

23.5. Spring configuration

Users of Spring Data Neo4j have two ways of very concisely configuring it. Either they can use a Spring Data Neo4j XML configuration namespace, or they can use a Java-based bean configuration.

23.5.1. XML namespace

The XML namespace can be used to configure Spring Data Neo4j. The `config` element provides an XML-based configuration of Spring Data Neo4j in one line. It has four attributes. * `base-package` points to a set of packages (provided as a comma separated String of names) which SDN will scan for locate all of your domain entity classes (`@NodeEntity` and `@RelationshipEntity`). NOTE: Neo4j 2.0 introduced the requirement to separately manage schema and data transactions which altered some options for SDN with regards be being able to automatically detect and register `@NodeEntity` and `@RelationshipEntity`s on the fly. Several approaches were attempted to try and handle this automatically with SDN 3.0.X, none of which worked in a satisfactory manner. This has resulted in the base-package becoming a mandatory field now with entity metadata handling becoming an explicit step in the lifecycle. * `graphDatabaseService` points out the Neo4j instance to use. * `storeDirectory` is a convenient alternative (instead of `graphDatabaseService`) to point to a directory where a new `EmbeddedGraphDatabase` will be created. * `entityManagerFactory` is only required for cross-store configuration.

Example 104. XML configuration with store directory

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/data/neo4j
            http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">

    <context:annotation-config/>
    <neo4j:config
           storeDirectory="target/config-test"
           base-package="org.example.domain"/>

</beans>
```

Example 105. XML configuration with basic GraphDatabaseService bean

```
<context:annotation-config/>

<bean id="graphDatabaseService" scope="singleton" destroy-method="shutdown"
      class="org.springframework.data.neo4j.support.GraphDatabaseServiceFactoryBean">
    <constructor-arg value="target/config-test"/>
</bean>

<neo4j:config graphDatabaseService="graphDatabaseService" base-package=
"org.example.domain"/>
```

Example 106. XML configuration with disabled index creation (e.g. for HA-slave)

```
<context:annotation-config/>
<neo4j:config
       storeDirectory="target/config-test"
       create-index="false"
       base-package="org.example.domain"/>
```

Example 107. XML configuration with cross-store

```
<context:annotation-config/>

<bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
        id="entityManagerFactory">
    <property name="dataSource" ref="dataSource"/>
    <property name="persistenceXmlLocation" value="classpath:META-
INF/persistence.xml"/>
</bean>

<neo4j:config storeDirectory="target/config-test"
        entityManagerFactory="entityManagerFactory"
        base-package="org.example.domain"/>
```

23.5.2. Repository Configuration

Spring Data Neo4j repositories are configured using the `<neo4j:repositories>` element which defines the base-package (or packages) for the repositories. A reference to an existing `Neo4jTemplate` bean reference can be passed in as well.

As Spring Data Neo4j repositories build upon the infrastructure provided by Spring Data Commons, the configuration options for repositories described there work here as well.

Example 108. XML configuration for repositories

```
<neo4j:repositories base-package="org.example.repository"/>
```

23.5.3. Java-based bean configuration

You can also configure Spring Data Neo4j using Java-based bean metadata.

NOTE For those not familiar with Java-based bean configuration in Spring, we recommend that you read up on it first. The Spring documentation has a high-level introduction as well as detailed documentation on it.

In order to configure Spring Data Neo4j with Java-based bean config, the class `Neo4jConfiguration` is registered with the context. This is either done explicitly in the context configuration, or via classpath scanning for classes that have the @Configuration annotation. The only thing that must be provided is the `GraphDatabaseService` and the `basePackage` must also be set. The examples below show how this can be done.

Example 109. Pure Java based bean configuration

```
@Configuration
@EnableNeo4jRepositories(basePackages = "org.example.repositories")
public class BasicJavaConfig extends Neo4jConfiguration {

    public BasicJavaConfig() {
        setBasePackage("org.example.domain");
    }

    @Bean
    public GraphDatabaseService graphDatabaseService() {
        return new GraphDatabaseFactory().newEmbeddedDatabase("path/to/mydb");
    }

    // You can add your own beans here, and/or override some of the
    // default config (such as Type Representation Strategies etc)

}
```

Example 110. Java-based bean config registration via XML

To register the default `@Configuration` `Neo4jConfiguration` class, as well as Spring's `ConfigurationClassPostProcessor` that transforms the `@Configuration` class to bean definitions via XML.

```
<beans ...>
    ...
    <tx:annotation-driven mode="aspectj" transaction-manager="transactionManager"/>
    <bean class="org.springframework.data.neo4j.config.Neo4jConfiguration">
        <property name="basePackage" value="org.example.domain" />
    </bean>

    <bean class=
"org.springframework.context.annotation.ConfigurationClassPostProcessor"/>

    <bean id="graphDatabaseService" class=
"org.springframework.data.neo4j.support.GraphDatabaseServiceFactoryBean"
          destroy-method="shutdown">
        <constructor-arg value="target/config-test"/>
    </bean>
    ...
</beans>
```

Additional beans can be configured to be included in the Neo4j-Configuration just by defining them in the Spring context. ConversionService for custom conversions, Validators for bean validation, TypeRepresentationStrategyFactory for configuring the in graph type representation, IndexProviders for custom index handling (e.g. for multi-tenancy) or Entity-Instantiators (with their config) to have more control over the creation of entity instances and much more.

Chapter 24. Cross-store persistence

The Spring Data Neo4j project support cross-store persistence for the advanced mapping mode, which allows for parts of the data to be stored in a traditional JPA data store (RDBMS), and other parts in a graph store. This means that an entity can be partially stored in e.g. MySQL, and partially stored in Neo4j.

This allows existing JPA-based applications to embrace NOSQL data stores for evolving certain parts of their data model. Possible use cases include adding social networking or geospatial information to existing applications.

24.1. Partial entities

Partial graph persistence is achieved by restricting the Spring Data Neo4j aspects to manage only explicitly annotated parts of the entity. Those fields will be made `@Transient` by the aspect so that JPA ignores them.

A backing node in the graph store is only created when the entity has been assigned a JPA ID. Only then will the association between the two stores be established. Until the entity has been persisted, its state is just kept inside the POJO (in detached state), and then flushed to the backing graph database on the persist operation.

The association between the two entities is maintained via a FOREIGN_ID field in the node, that contains the JPA ID. Currently only single-value IDs are supported. The entity class can be resolved via the `TypeRepresentationStrategy` that manages the Java type hierarchy within the graph database. Given the ID and class, you can then retrieve the appropriate JPA entity for a given node.

The other direction is handled by indexing the Node with the FOREIGN_ID index which contains a concatenation of the fully qualified class name of the JPA entity and the ID. The matching node can then be found using the indexing facilities, and the two entities can be reassociated.

Using these mechanisms and the Spring Data Neo4j aspects, a single POJO can contain some fields handled by JPA and others handles by Spring Data Neo4j. This also includes relationship fields persisted in the graph database.

24.2. Cross-store annotations

Cross-store persistence only requires the use of one additional annotation: `@GraphProperty`. See below for details and an example.

24.2.1. @NodeEntity(partial = "true")

When annotating an entity with `partial = true`, this marks it as a cross-store entity. Spring Data Neo4j will thus only manage fields explicitly annotated with `@GraphProperty`.

24.2.2. @GraphProperty

Fields of primitive or convertible types do not normally have to be annotated in order to be persisted by Spring Data Neo4j. In cross-store mode, Spring Data Neo4j **only** persists fields explicitly annotated with `@GraphProperty`. JPA will ignore these fields.

24.2.3. Example

The following example is taken from the Spring Data Neo4j examples myrestaurants-social project:

Example 111. Cross-store node entity

```
@Entity
@Table(name = "user_account")
@NodeEntity(partial = true)
public class UserAccount {
    private String userName;
    private String firstName;
    private String lastName;

    @GraphProperty
    String nickname;

    @RelatedTo
    Set<UserAccount> friends;

    @RelatedToVia(type = "recommends")
    Iterable<Recommendation> recommendations;

    @Temporal(TemporalType.TIMESTAMP)
    @DateTimeFormat(style = "S-")
    private Date birthDate;

    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Restaurant> favorites;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;

    public void knows(UserAccount friend) {
        relateTo(friend, "friends");
    }

    public Recommendation rate(Restaurant restaurant, int stars, String comment) {
        Recommendation recommendation = relateTo(restaurant, Recommendation.class,
"recommends");
        recommendation.rate(stars, comment);
        return recommendation;
    }

    public Iterable<Recommendation> getRecommendations() {
        return recommendations;
    }
}
```

24.3. Configuring cross-store persistence

Configuring cross-store persistence is done similarly to the default Spring Data Neo4j configuration. All you need to do is to specify an `entityManagerFactory` in the XML namespace `config` element, and Spring Data Neo4j will configure itself for cross-store use.

Example 112. Cross-store Spring configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:datagraph="http://www.springframework.org/schema/data/neo4j"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/data/neo4j
        http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd
        ">

    <context:annotation-config/>

    <neo4j:config storeDirectory="target/config-test"
        entityManagerFactory="entityManagerFactory"/>

    <bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
            id="entityManagerFactory">
        <property name="dataSource" ref="dataSource"/>
        <property name="persistenceXmlLocation" value="classpath:META-
INF/persistence.xml"/>
    </bean>
</beans>
```

Chapter 25. Sample code

25.1. Introduction

Spring Data Neo4j comes with a number of sample applications. The source code of the samples can be found on Github. The different sample projects are introduced below.

25.2. Hello Worlds sample application

The Hello Worlds sample application exists merely to provide a client with a way of creating some "worlds" (node entities) and "rocket routes" (relationships) between worlds, all in a galaxy (the graph). There is currently no GUI for this application, rather you can have a look at the associated unit tests for some examples of how to interact with this domain via a dedicated `GalaxyService` class.

The unit tests additionally demonstrate some other features of Spring Data Neo4j as well. The sample comes with a minimal configuration for Maven and Spring to get up and running quickly.

The Hello Worlds application is available both for the simple mapping (`hello-worlds`) and for the advanced mapping (`hello-world-aspects`).

Running the unit tests create two versions of the world, one based on the default Label based Type Representation Strategy, and another (for comparison purposes) based on the Sub Reference type representation strategy. From the `GalaxyService` clients perspective, there is no difference, however how the data physically stored in the graph, does differ, and we take this opportunity to show you how using two different type representation strategies can impact how your data is modelled in the graph itself.

Hello World Galaxy - Using default "Label" Type Representation Strategy

Hello World Galaxy - Using "Sub Reference" Type Representation Strategy

25.3. IMDB sample application

NOTE This sample application still needs to be upgraded to SDN 3.0.X

```
The IMDB sample is a web application that imports datasets from the Internet Movie
Database (IMDB) into the graph database. It allows the listing of movies with their
actors, and of actors and their roles in different movies. It also uses graph traversal
operations to calculate the http://en.wikipedia.org/wiki/Bacon_number[Bacon number] of
any given actor. This sample application shows the usage of Spring Data Neo4j in a more
complex setting, using several annotated entities and relationships as well as indexes
and in-graph indexes and graph traversals.
```

See the readme file for instructions on how to compile and run the application.

An excerpt of the data stored in the graph database after executing the application:

25.4. MyRestaurants sample application

NOTE This sample application still needs to be upgraded to SDN 3.0.X

Simple, JPA-based web application for managing users and restaurants, with the ability to add restaurants as favorites to a user. It is basically the foundation for the MyRestaurants-Social application (see[samples:myrestaurants-social]), and does therefore not use Spring Data Neo4j.

25.5. MyRestaurant-Social sample application

NOTE This sample application still needs to be upgraded to SDN 3.0.X

This application extends the MyRestaurants sample application, adding social networking functionality to it with cross-store persistence. The web application allows for users to add friends and rate restaurants. A graph traversal provides recommendations based on your friends' (and their friends') rating of restaurants.

Here's an excerpt of the data stored in the graph database after executing the application:

25.6. Cineasts social movie database

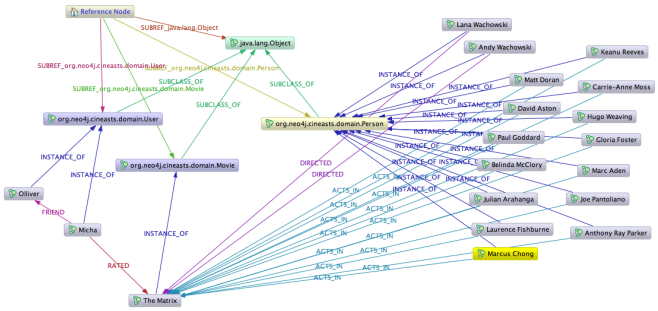
NOTE This sample application still needs to be upgraded to SDN 3.0.X

The cineasts.net application was introduced extensively in the first part of this guide, the tutorial. The tutorial covers the development of the simple mapping version of cineasts.

To document the differences, versions for the advanced mapping (`cineasts-aspects`) and accessing the remote server (`cineasts-rest`) are also available.

A online version of cineasts can be found on cineasts.net. A sample dataset of the cineasts databse is available at the neo4j sample-data page.

This is a subset of the visualization of the cineasts graph for the "Matrix" movie.




Login Register
CINEASTS
WELCOME TO CINEASTS.NET
Find movie
This site is running on SpringFramework and Spring Data Graph powered by the Neo4j graph database. All movie data is provided by themoviedb.org.
springsource
A division of vmware
springdata
springdatagraph
Neo4j
the graph database

Chapter 26. Heroku: Seeding the Cloud

Deploying your application into the cloud is a great way to scale from from "wouldn't it be cool if.." to giving interviews to Forbes, Fast Company, and Jimmy Fallon. Heroku makes it super easy to provision everying you need, including a Neo4j Add-on. With a few simple adjustments, your Spring Data Neo4j application is ready to take that first step into the cloud.

To deploy your Spring Data Neo4j web application to Heroku, you'll need:

- account on Heroku
- git command line
- maven-based project
- standard Spring MV Servlet application
- well, and Spring Data Neo4j REST

For reference, the following sections detail the steps taken to make the Spring Data Neo4j Todos example ready for deployment to Heroku.

26.1. Create a Self-Hosted Web Application

Usually, a Spring MVC application is bundled into a war and deployed to an application server like Tomcat. But Heroku can host any kind of java application. It just needs to know what to launch. So, we'll transform the war into a self-hosted servlet using an embedded Jetty server, then add a startup script to launch it.

First, we'll add the dependencies for Jetty to the `pom.xml`:

Example 113. Jetty dependencies - pom.xml

```
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-webapp</artifactId>
    <version>7.4.4.v20110707</version>
</dependency>
<dependency>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>jsp-2.1-glassfish</artifactId>
    <version>2.1.v20100127</version>
</dependency>
```

Then we'll change the scope of the servlet-api artifact from `provided` to `compile`. This library is normally provided at runtime by the application container. Since we're self-hosting, it needs to be included directly. Make sure the servlet-api dependency looks like this:

Example 114. servlet-api dependencies - pom.xml

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>compile</scope>
</dependency>
```

We could provide a complicated command-line to Heroku to launch the app. Instead, we'll simplify the command-line by using the `appassembler-maven-plugin` to create a launch script. Add the plugin to your pom's `build/plugins` section:

Example 115. appassembler-maven-plugin configuration pom.xml

```
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>appassembler-maven-plugin</artifactId>
<version>1.1.1</version>
<executions>
  <execution>
    <phase>package</phase>
    <goals><goal>assemble</goal></goals>
    <configuration>
      <assembleDirectory>target</assembleDirectory>
      <extraJvmArguments>-Xmx512m</extraJvmArguments>
      <programs>
        <program>
          <mainClass>Main</mainClass>
          <name>webapp</name>
        </program>
      </programs>
    </configuration>
  </execution>
</executions>
</plugin>
```

Finally, switch the packaging from `war` to `jar`. That's it for the pom.

Now that the application is ready to be self-hosted, create a simple `Main` to bootstrap Jetty and host the servlet.

Example 116. src/main/java/Main.java

```
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.webapp.WebAppContext;
public class Main {
    public static void main(String[] args) throws Exception {
        String webappDirLocation = "src/main/webapp/";
        String webPort = System.getenv("PORT");
        if(webPort == null || webPort.isEmpty()) {
            webPort = "8080";
        }
        Server server = new Server(Integer.valueOf(webPort));
        WebAppContext root = new WebAppContext();
        root.setContextPath("/");
        root.setDescriptor(webappDirLocation+"/WEB-INF/web.xml");
        root.setResourceBase(webappDirLocation);
        root.setParentLoaderPriority(true);
        server.setHandler(root);
        server.start();
        server.join();
    }
}
```

Notice the use of environment variable "PORT" for discovering which port to use. Heroku and the Neo4j Add-on use a number of environment variable to configure the application. Next, we'll modify the Spring application context to use the Neo4j variables for specifying the connection to Neo4j itself.

In the SDN Todos example, `src/main/resources/META-INF/spring/applicationContext-graph.xml` was modified to look like this:

Example 117. Spring Data Neo4j REST configuration - applicationContext-graph.xml

```
<neo4j:config graphDatabaseService="graphDatabaseService"/>
<bean id="graphDatabaseService"
    class="org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
    <constructor-arg index="0" value="${NEO4J_REST_URL}" />
    <constructor-arg index="1" value="${NEO4J_LOGIN}" />
    <constructor-arg index="2" value="${NEO4J_PASSWORD}" />
</bean>
```

Before provisioning at Heroku, test the application locally. First make sure you've got Neo4j server running locally, using default configuration. Then set the following environment variables:

Example 118. environment variables

```
export NEO4J_REST_URL=http://localhost:7474/db/data
export NEO4J_LOGIN=""
export NEO4J_PASSWORD=""
```

Now you can launch the app by running `sh target/bin/webapp`. If running the SDN Todos example, you can test it by running `./bin/todos list`. That should return an empty JSON array, since no todos have been created yet.

For details about the `todos` script, see the `readme` included with the example.

26.2. Deploy to Heroku

With a self-hosted application ready, deploying to Heroku needs a few more steps. First, create a `Procfile` at the top-level of the project, which will contain a single line identifying the command line which launches the application.

The contents of the `Procfile` should contain:

Example 119. Procfile

```
web: sh target/bin/webapp
```

Example 120. deploy to heroku

```
# Initialize a local git repository, adding all the project files
  git init
  git add .
  git commit -m "initial commit"

# Provision a Heroku stack, add the Neo4j Add-on and deploy the appication

  heroku create --stack cedar
  heroku addons:add neo4j
  git push heroku master
```

NOTE	Note that the stack must be "cedar" to support running Java. Check that the process is running by using `heroku ps`, which should show a "web.1" process in the "up" state. Success!

For the SDN Todos application, you can try out the remote application using the `-r` switch with the `bin/todo` script like this:

Example 121. Session with todo script

```
./bin/todo -r mk "tweet thanks for the good work @mesirii @akollegger"
./bin/todo -r list
```

To see the Neo4j graph you just created through Heroku, use `heroku config` to reveal the `NEO4J_URL` environment variable, which will take you to Neo4j's Webadmin.

Chapter 27. Performance considerations

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

27.1. When to use Spring Data Neo4j

The focus of Spring Data Neo4j is to add a convenience layer on top of the Neo4j API. This enables developers to get up and running with a graph database very quickly, having their domain objects mapped to the graph with very little work. Building on this foundation, one can later explore other, more efficient ways to explore and process the graph - if the performance requirements demand it.

Like with any other object mapping framework, the domain entities that are created, read, or persisted 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.

Spring Data Neo4j, however, was not designed with a major focus on performance. It does add some overhead to pure graph operations.

Most of the overhead comes from the use of the Java Reflection API, which is used to provide information about annotations, fields and constructors. Some of the information is already cached by the JVM and the library infrastructure from Spring-Data-Commons, so that only the first access gets a performance penalty. Other reflection penalties like field or method access will occur all the time.

For the **simple mapping** it is important to be aware of the size graph of data that is pulled out of the graph database in a single read and copied to domain entities. That's why Spring Data Neo4j loads related data not by default. You have to provide an indicator (`@Fetch`) to do so. Alternatively the `Neo4jTemplate.fetch` method offers means of of loading entities and collections of those.

For the **advanced mapping mode** keep in mind that any access of properties and relationships will in general read through down to the database. To avoid multiple reads, it is sensible to store the result in a local variable in suitable scope (e.g. method, class or jsp).

To evaluate if the performance of Spring Data Neo4j impacts a certain use-case it is sensible to define performance requirements and measure the actual time in realistic test scenarios for the use-case. Only if Spring Data Neo4j doesn't perform as fast as required it is recommended to drop down to the native Neo4j API.

Chapter 28. AspectJ details

The advanced mapping mode of Spring Data Neo4j relies heavily on AspectJ. AspectJ is a Java implementation of the aspect-oriented programming paradigm that allows easy extraction and controlled application of so-called cross-cutting concerns. Cross-cutting concerns are typically repetitive tasks in a system (e.g. logging, security, auditing, caching, transaction scoping) that are difficult to extract using the normal OO paradigms. Many OO concepts, such as subclassing, polymorphism, overriding and delegation are still cumbersome to use with many of those concerns applied in the code base. Also, the flexibility becomes limited, potentially adding quite a number of configuration options or parameters.

The AspectJ pointcut language can be intimidating, but a developer using Spring Data Neo4j will not have to deal with that. Users don't have care about hooking into a framework mechanism, or having to extend a framework superclass.

AspectJ uses a declarative approach, defining concrete "advice", which is just pieces of code that contain the implementation of the "concern", as it is called. An AspectJ advice can for instance be applied before, after, or instead of a method or constructor call. It can also be applied on variable and field access. This is declared using AspectJ's expressive pointcut language, which is able to express any place within a code structure or flow. AspectJ is also able to introduce new methods, fields, annotations, interfaces, and superclasses to existing classes.

Spring Data Neo4j uses a mix of these mechanisms internally. First, when encountering the `@NodeEntity` or `@RelationshipEntity` annotations it introduces a new interface `NodeBacked` or `RelationshipBacked` to the annotated class. Secondly, it introduces fields and methods to the annotated class. See Active Record Methods for Advanced Mapping Mode for more information on the methods introduced.

Spring Data Neo4j also leverages AspectJ to intercept access to fields, delegating the calls to the graph database instead. Under the hood, properties and relationships will be created.

So how is an aspect applied to a concrete class? At compile time, the AspectJ Java compiler (ajc) takes source files and aspect definitions, and compiles the source files while adding all the necessary interception code for the aspects to hook in where they're declared to. This is known as compile-time **weaving**. At runtime only a small AspectJ runtime is needed, as the byte code of the classes has already been rewritten to delegate the appropriate calls via the declared advice in the aspects.

NOTE	A caveat of using compile-time weaving is that all source files that should be part of the weaving process must be compiled with the AspectJ compiler. Fortunately, this is all taken care of seamlessly by the AspectJ Maven plugin.

AspectJ also supports other types of weaving, e.g. load-time weaving and runtime weaving. These are currently not supported by Spring Data Neo4j.

Chapter 29. Neo4j Server

Neo4j is not only available in embedded mode. It can also be installed and run as a stand-alone server accessible via a HTTP API. Developers can integrate Spring Data Neo4j into the Neo4j server infrastructure in two ways: as a server extension, or remotely via the HTTP API.

Spring Data Neo4j was historically built around the Neo4j embedded Java APIs, but are not optimized for remote usage. Most of the Graph Database operations are sent as Cypher statements to the server's transactional Cypher endpoint. Only the few operations that are not supported by Cypher yet (legacy indexes, traversals, management operations) use the Neo4j Server REST API.

29.1. Server Extension

When would you write a server extension? If you want to achieve the performance, that you get from the embedded Neo4j usage in Spring Data Neo4j, then a server extension is the easiest way. Running as a extension within Neo4j Server, Spring Data Neo4j can access the Neo4j the same way as running with a embedded database inside your Spring application.

The Neo4j Server has two built-in extension mechanisms. It is possible to extend existing REST endpoints for the graph database, nodes, or relationships, adding new service URIs or methods to those. This is achieved by writing a server plugin. However this approach has some restrictions in terms of HTTP verbs and result types.

For an unrestricted implementation, an unmanaged extension can be used. Unmanaged extensions are essentially Jersey resource implementations. The resource constructors or methods can get `@Context GraphDatabaseService` and `@Context CypherExecutor` instances injected to run the necessary Neo4j API calls and Cypher statements and return appropriate `Representations`.

Both kinds of extensions have to be packaged as JAR files and added to the Neo4j Server's plugin directory. Server Plugins are picked up by the server at startup if they provide the necessary `META-INF.services/org.neo4j.server.plugins.ServerPlugin` file for Java's ServiceLoader facility. Unmanaged extensions have to be registered with the Neo4j Server configuration in `conf/neo4j-wrapper.conf`.

Example 122. Configuring an unmanaged extension

```
org.neo4j.server.thirdparty_jaxrs_classes=com.example.mypackage=/my-context
```

Integrating Spring Data Neo4j ApplicationContext configuration in the Neo4j Server is easy. You provide the Spring context configuration location, and list which Spring-beans should be exposed:

Example 123. Server plugin initialization

```
public class HelloWorldInitializer extends SpringPluginInitializer {
    public HelloWorldInitializer() {
        super(new String[]{"spring/helloWorldServer-Context.xml"},
              Pair.of("worldRepository", WorldRepository.class),
              Pair.of("template", Neo4jTemplate.class));
    }
}
```

Now, your resources can require the Spring beans they need as parameters, annotated with `@Context`:

Example 124. Jersey resource

```
@Path( "/path" )
@POST
@Produces( MediaType.APPLICATION_JSON )
public void foo( @Context WorldRepository repo ) {
    ...
}
```

The `SpringPluginInitializer` merges the server provided `GraphDatabaseService` with the Spring configuration and registers the named beans as Jersey `Injectables`. It is still necessary to list the `SpringPluginInitializer` implementation's fully qualified class name in a file named `META-INF/services/org.neo4j.server.plugins.PluginLifecycle`, e.g. `org.example.extension.HelloWorldInitializer`. The Neo4j Server can then pick up and run the initialization classes before the extensions are loaded.

29.2. Using Spring Data Neo4j as a Neo4j Server client

To use Neo4j's remote APIs, you can use them directly to send Cypher statements to the server, e.g. with the Neo4j-JDBC driver. That JDBC driver integrates well with the commonly used `spring-jdbc` libraries and classes.

There are also other remote drivers for Neo4j available.

Spring Data Neo4j's integration with the server also uses the Cypher endpoint to execute GraphDatabaseService operations transactionally against the server. The implementation of the integration is handled by `SpringCypherRestGraphDatabase` and `RestAPICypherImpl` which wraps the *old* REST-API methods in the appropriate Cypher statement calls. It integrates with the Spring Transaction APIs by providing a `javax.transaction.TransactionManager` implementation that is configured to be used by the `JtaTransactionManager` bean provided by Spring Data Neo4j.

By simply configuring the `graphDatabaseService` to be a `SpringCypherRestGraphDatabase` pointing to a Neo4j Server instance and referring to that from `<neo4j:config>`, Spring Data Neo4j will use the server side database for both the simple mapping as well as the advanced mapping.

NOTE	The Neo4j Server REST API does not allow for transactions to span across requests, which means that all operations that are not handled by Cypher (traversals, legacy index lookups and management operations) are not participating in the Cypher transactions.

Please also keep in mind that performing graph operations via the remote API is slower than local operations. You have to take roundtrip latency, request serialization and result parsing into account. Also remember that Spring Data Neo4j was built around Neo4j's embedded APIs, that's why it is not the most efficient user of the remote API. A new version of a Java OGM and Spring Data Neo4j is in development that addresses these issues.

If the mapping CRUD operations are too slow, try to avoid the automatic fetching of additional levels of entities. Basic CRUD should be fast enough. Use Cypher to execute operations within the server and map the results using `@QueryResult` POJOs or interfaces.

To set up your project to use the remote Neo4j Server integration, add this dependency to your `pom.xml`:

Example 125. Remote Client configuration - pom.xml

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j-rest</artifactId>
  <version>3.3.0.M1</version>
</dependency>
```

Now, you set up the normal Spring Data Neo4j configuration, but point the database instance to an URL instead of a local directory:

Example 126. Remote configuration - application context

```
<neo4j:config graphDatabaseService="graphDatabaseService"/>

<bean id="graphDatabaseService" class=
"org.springframework.data.neo4j.rest.SpringCypherRestGraphDatabase">
    <constructor-arg value="http://localhost:7474/db/data/" index="0"/>
<!-- for running against a server requiring authentication
    <constructor-arg value="username" index="1"/>
    <constructor-arg value="password" index="2"/>
-->
</bean>
```

Your project is now set up to work with a remote Neo4j Server.

For direct execution of Cypher graph queries and graph traversals it is sensible to forward those to the remote side and execute them on the server. `SpringCypherRestGraphDatabase` already supports this approach by providing appropriate methods. (e.g. `query()`, `queryEngineFor()`, `index()` and `createTraversalDescription()`). Please use those methods when interacting with a remote server for better performance. Those methods are also used by the Neo4jTemplate and the mapping infrastructure implementation.

Appendix

Appendix A: Namespace reference

The <repositories /> element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces. [see [repositories.create-instances.spring]]

Table 1. Attributes

Name	Description
`base-package`	Defines the package to be used to be scanned for repository interfaces extending *Repository (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to `Impl`.
`query-lookup-strategy`	Determines the strategy to be used to create finder queries. See [repositories.query-methods.query-lookup-strategies] for details. Defaults to `create-if-not-found`.
`named-queries-location`	Defines the location to look for a Properties file containing externally defined queries.
`consider-nested-repositories`	Controls whether nested repository interface definitions should be considered. Defaults to `false`.

Appendix B: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure. [see [repositories.create-instances.spring]]

Table 2. Attributes

Name	Description
`locations`	Where to find the files to read the objects from the repository shall be populated with.

Appendix C: Repository query keywords

Supported query keywords

The following table lists the keywords generally supported by the Spring Data repository query derivation mechanism. However, consult the store-specific documentation for the exact list of supported keywords, because some listed here might not be supported in a particular store.

Table 3. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_NOT_NULL`	`NotNull`, `IsNotNull`
`IS_NULL`	`Null`, `IsNull`
`LESS_THAN`	`LessThan`, `IsLessThan`
`LESS_THAN_EQUAL`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`NEAR`	`Near`, `IsNear`
`NOT`	`Not`, `IsNot`
`NOT_IN`	`NotIn`, `IsNotIn`
`NOT_LIKE`	`NotLike`, `IsNotLike`

Logical keyword	Keyword expressions
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith
TRUE	True, IsTrue
WITHIN	Within, IsWithin

Appendix D: Repository query return types

Supported query return types

The following table lists the return types generally supported by Spring Data repositories. However, consult the store-specific documentation for the exact list of supported return types, because some listed here might not be supported in a particular store.

NOTE Geospatial types like (`GeoResult`, `GeoResults`, `GeoPage`) are only available for data stores that support geospatial queries.

Table 4. Query return types

Return type	Description
`void`	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	An unique entity. Expects the query method to return one result at most. In case no result is found `null` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. In case no result is found `Optional.empty()`/`Optional.absent()` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Stream<T>`	A Java 8 `Stream`.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, e.g. distance to a reference location.
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, e.g. average distance to a reference location.