Good Relationships

The Spring Data Graph Guide Book

1.1.0.M1

Foreword by Rod Johnson

I’m excited about Spring Data Graph 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 Graph 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 Graph 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 Graph 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 Graph — 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 Graph 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 Graph book, and happy coding!

Rod Johnson, Founder, Spring and SVP, Application Platform, VMware

Foreword by Emil Eifrem

"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 Graph 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 Graph 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 Graph guide book and welcome to the wonderful world of graph databases!

Emil Eifrem, CEO of Neo Technology

About this guide book

Welcome to the Spring Data Graph Guide Book. Thank you for taking the time to get an in depth look into Spring Data Graph. 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 Graph, as the name alludes to, aims to provide support for graph databases. It currently supports Neo4j.

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

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

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 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 Graph 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 Graph, it can of course also be read cover-to-cover.

Enjoy the book!

Part I. 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 Graph and Neo4j. Cineasts are people who love movies, and the site is a gathering place for these people. 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 Graph. The complete source code for the app is available on Github.

Chapter 1. Introducing our project

Allow me to introduce Cineasts.net

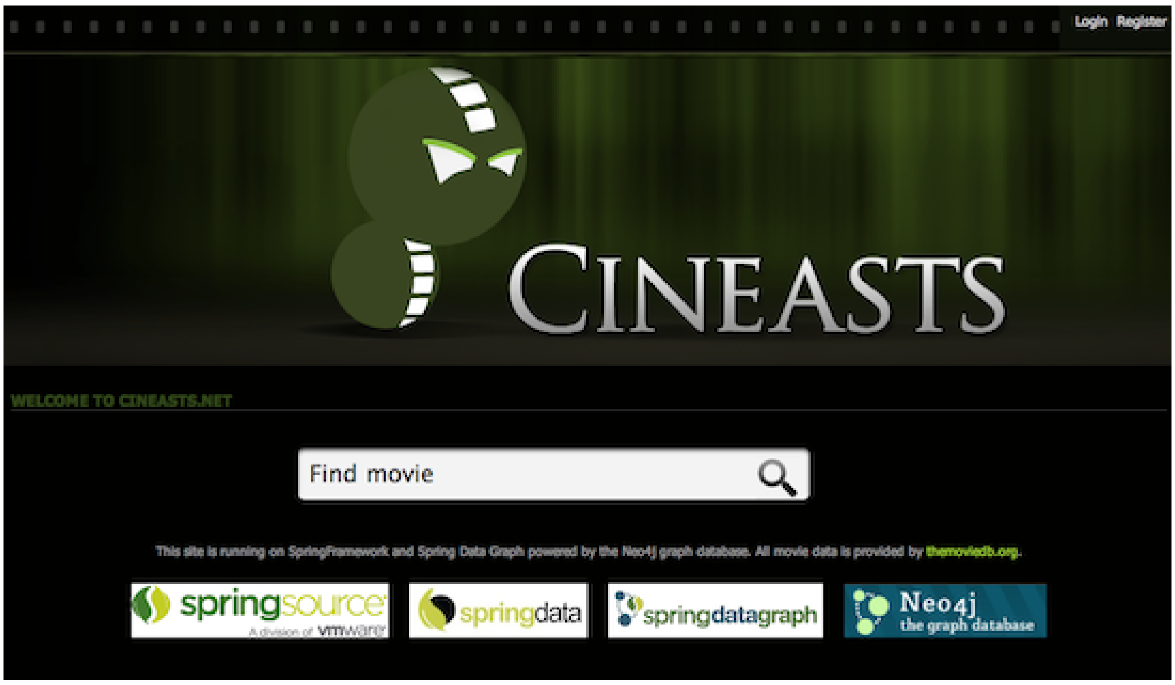
Once upon a time we wanted to build a social movie database. At first there was only the name: Cineasts, the cinema enthusiasts who have a burning passion for movies. So we went ahead and bought the domain cineasts.net, and the project was almost done.

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.

CINEASTS

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:



This site is running on SpringFramework and Spring Data Graph powered by the Neo4j graph database. All movie data is provided by themoviedb.org.

Chapter 2. 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 Graph for Neo4j, a graph database. Neo4j's slogan of "value in relationships" and the accompanying docs looked like what we needed. We decided to give it a try.

2.1. 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 2.1. Project pom.xml

```
<properties>
    <spring.version>3.0.5.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>

<build><plugins>
 <plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>7.1.2.v20100523</version>
  <configuration>
     <webAppConfig>
       <contextPath>/</contextPath>
     </webAppConfig>
  </configuration>
 </plugin>
</plugins></build>
```

Example 2.2. 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 2.3. Project 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="aspectj"/>
</beans>
```

Example 2.4. Project dispatcherServlet-servlet.xml

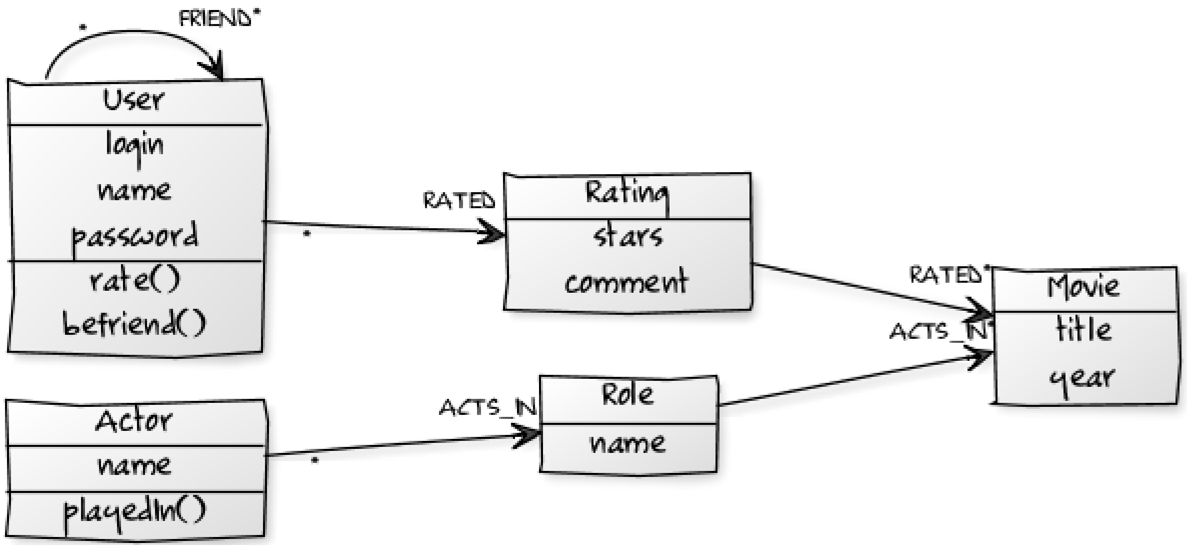
```
<mvc:annotation-driven/>
<mvc:resources mapping="/images/**" location="/images/"/>
<mvc:resources mapping="/resources/**" location="/resources/"/>
<context:component-scan base-package="org.neo4j.cineasts.controller"/>

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver" p:pr

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

We spun up Jetty by doing `mvn jetty:run` to see if there were any obvious issues with the config. It all seemed to work just fine.

Chapter 3. 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 3.1. Domain model

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

class Actor {
    int 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 tests to show how the basic plumbing works.

Chapter 4. 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. Relationships are first-class citizens in Neo4j, meaning 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.

We also learned that Neo4j is fully transactional and therefore upholds ACID guarantees for our data. 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 for later.

We started out by doing some prototyping with the Neo4j core API to get a feeling for that. 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 4.1. Neo4j Maven dependency

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>1.3.M05</version>
</dependency>
```

Example 4.2. 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("Tom Hanks");

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

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

Chapter 5. Spring Data Graph

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 Graph promised to do the heavy lifting for us, so we continued with investigating it.

Spring Data Graph depends heavily on AspectJ weaving. Some parts of our classes would get new behavior, but it would not be visible in our code. The upside of this is that you get rid of a lot of boilerplate code.

The first step was to configure Maven:

Example 5.1. Spring Data Graph Maven configuration

```
<properties>
    <aspectj.version>1.6.11.RELEASE</aspectj.version>
</properties>

<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j</artifactId>
  <version>1.0.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${aspectj.version}</version>
</dependency>

<build> <plugins> <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.2</version>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</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</artifactId>
            </aspectLibrary>
        </aspectLibraries>
        <source>1.6</source>
        <target>1.6</target>
    </configuration>
</plugin> </plugins> </build>
```

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

Example 5.2. Spring Data Graph context configuration

```
<beans xmlns="http://www.springframework.org/schema/beans" ...
       xmlns:datagraph="http://www.springframework.org/schema/data/graph"
       xsi:schemaLocation="... http://www.springframework.org/schema/data/graph
       http://www.springframework.org/schema/data/graph/datagraph-1.0.xsd">
    ...
    <datagraph:config storeDirectory="data/graph.db"/>
    ...
</beans>
```

Chapter 6. Annotating the domain

Decorations

Looking at the Spring Data Graph documentation, we found a simple Hello World example and tried to understand it. 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.

It was time to put our entities to a test. How could we now be assured that a attribute really was persisted to the graph store? We wanted to load the entity and check the attribute. Either we could have a GraphDatabaseContext injected and use its `getById(entityId)` method to load the entity. Or use a more versatile Repository. We decided to keep things simple for now. Looking at the documentation revealed that there are a bunch of methods introduced to the entities by the aspects to support working with the entities. That's not entirely obvious. We found two that would do the job: `entity.persist()` `entity.getNodeId()`.

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

Example 6.1. First test case

```
@Autowired GraphDatabaseContext graphDatabaseContext;

@Test public void persistedMovieShouldBeRetrievableFromGraphDb() {
    Movie forrestGump = new Movie("Forrest Gump", 1994).persist();
    Movie retrievedMovie = graphDatabaseContext.getById(forrestGump.getNodeId());
    assertEqual("retrieved movie matches persisted one", forrestGump, retrievedMovie);
    assertEqual("retrieved movie title matches", "Forrest Gump", retrievedMovie.getTitle());
}
```

It worked! But hold on, what about transactions? After all, we had not declared the test to be transactional. After some further reading we learned that calling `persist()` outside of a transaction automatically creates an implicit transaction. Very much like an EntityManager would behave. We also learned that when performing more complex operations on the entities we'd need external transactions, but not for this simple test.

Our domain model had now evolved.

Example 6.2. Movie class

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

Chapter 7. 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. This time we went with the default GraphRepository (previously Finder) to retrieve the indexed movie.

Example 7.1. Exact Indexing for Movie id

```
@NodeEntity class Movie {
    @Indexed int id;
    String title;
    int year;
}

@Autowired DirectGraphRepositoryFactory graphRepositoryFactory;

@Test public void persistedMovieShouldBeRetrievableFromGraphDb() {
    int id = 1;
    Movie forrestGump = new Movie(id, "Forrest Gump", 1994).persist();
    GraphRepository<Movie> movieRepository = graphRepositoryFactory.createGraphRepository(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 8. Repositories

Serving a good cause

We wanted to add repositories with domain-specific operations. We started by creating a movie-specific repository, simply by creating an empty interface. It is more convenient to work with a named interface rather than different versions of a generic one.

Example 8.1. Movie repository

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

Then we added it to the Spring context configuration by simply adding:

Example 8.2. Repository context configuration

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

We then created the domain-specific repository class, annotating it with `@Repository` and `@Transactional`, and injected the movie repository.

Example 8.3. Domain-specific repository

```
@Repository @Transactional
public class CineastsRepostory {
  @Autowired MovieRepository movieRepository;

  public Movie getMovie(int id) {
      return movieRepository.findByPropertyValue("id", id);
  }
}
```

We did the same for the actors and users.

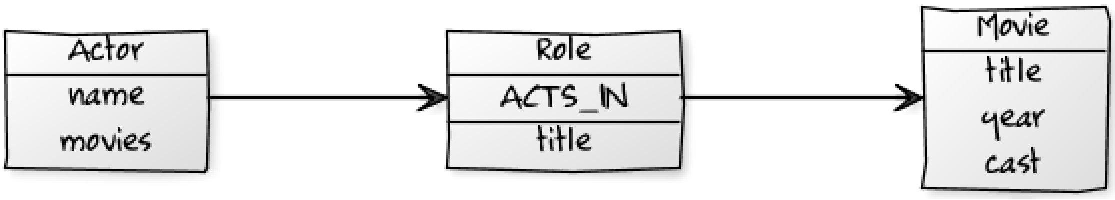
Chapter 9. Relationships

A convincing act

Our application was not yet very much fun, 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 assigned properties. That allows for representing them as entities if needed.

9.1. 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 9.1. Role class

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

When writing a test for that we tried to create the relationship entity with the `new` keyword, but we got an exception saying that it was not allowed. At first this surprised us, but then we realized that a relationship entity must have a starting entity and ending entity. It turned out that the aspect had introduced a `entity.relateTo` method in the node entities. It turned out to be exactly what we needed. We simply added a method to the Actor class, connecting it to movies.

Example 9.2. Relating actors to movies

```
class Actor {
...
    public Role playedIn(Movie movie, String roleName) {
        Role role = relateTo(movie, Role.class, "ACTS_IN");
        role.setRole(roleName);
        return role;
    }
}
```

9.2. Accessing related entities

Now we wanted to find connected entities. We already had fields for the relationships in both classes. Now it was time to annotate them correctly. It turned out that we needed to provide the target type of the fields again, due to Java's type erasure. 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.

Example 9.3. @RelatedTo usage

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

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

    public Role playedIn(Movie movie, String roleName) {
        Role role = relateTo(movie, Role.class, "ACTS_IN");
        role.setRole(roleName);
        return role;
    }
}
```

While reading about these relationship collections, we learned that they are actually Spring Data Graph-managed sets. So whenever we add or remove something from the set, it automatically gets reflected in the underlying relationships. That's neat! But this also meant we did not need to initialize the fields. That could be easy to forget.

We made sure to add a test for those, so we were assured that the collections worked as advertised.

9.3. Accessing the relationship entities

But we still couldn't access the Role relationships. It turned out that there was a separate annotation `@RelatedToVia` for accessing the actual relationship entities . And we had to declare the field as an Iterable<Role>, with read-only semantics. This appeared to mean that we were not able to add new roles though the field. Adding relationship entities seemed like it had to be done by using `entity.relateTo()`. The annotation attributes were similar to those used for `@RelatedTo`. So off we went, creating our first real relationship (just kidding).

Example 9.4. @RelatedToVia usage

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

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

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

Chapter 10. Get it running

Curtains up!

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

10.1. 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. To make it safe to call several times we added index lookups to check for existing entries. A simple /populate endpoint for the controller that called it would be enough for now.

Example 10.1. Populating the database - Controller

```
@Service
public class DatabasePopulator {

    @Autowired GraphDatabaseContext ctx;
    @Autowired CineastsRepository repository;

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

@Controller
public class MovieController {

    private DatabasePopulator populator;

    @Autowired
    public MovieController(DatabasePopulator populator) {
        this.populator = populator;
    }

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

Example 10.2. Populating the database - JSP

```
<%@ 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}</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="${rol
      </li>
    </c:forEach>
    </ul>
    </c:if>
  </c:when>
  <c:otherwise>
      No Movie with id ${id} found!
  </c:otherwise>
</c:choose>
```

Accessing the URI showed the single added movie on screen.

Note

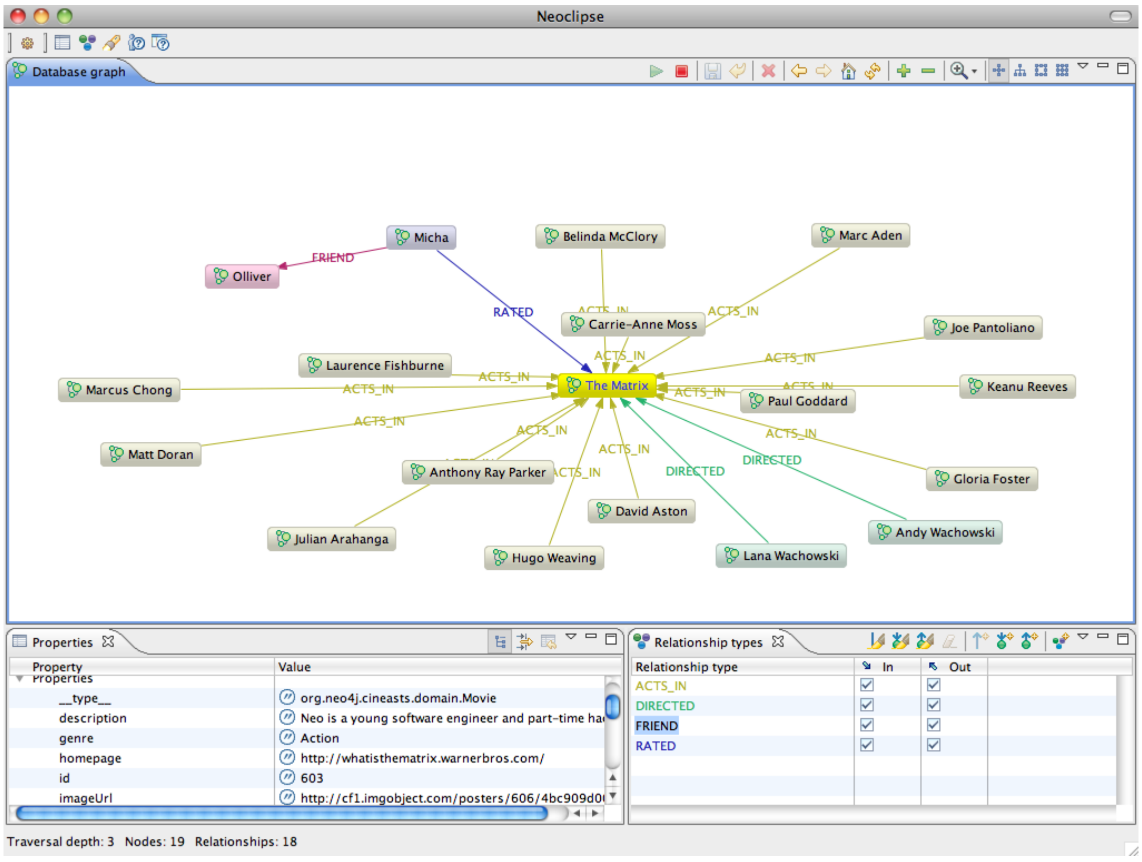
Pardon the misused GET parameter for that (don't try this at home, the REST guys will be hunt you down). This is only for running it from the browser address line. The next iteration of this website would use a button with POST.

10.2. 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.

10.2.1. Neoclipse visualization

First we tried Neoclipse, an Eclipse RCP application/plugin 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.



Property	Value
▼ Properties	
__type__	org.neo4j.cineasts.domain.Movie
description	Neo is a young software engineer and part-time ha
genre	Action
homepage	http://whatisthematrix.warnerbros.com/
id	603
imageUrl	http://cf1.imgobject.com/posters/606/4bc909d0

Relationship type	In	Out
ACTS_IN	✓	✓
DIRECTED	✓	✓
FRIEND	✓	✓
RATED	✓	✓

Traversal depth: 3 Nodes: 19 Relationships: 18

10.2.2. 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 enable_remote_shell=true), or directly open an existing graph store.

Example 10.3. Starting the Neo4j Shell

```
neo4j-shell -readonly -path data/graph.db
```

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 and traversals.

Example 10.4. 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 11. Web views

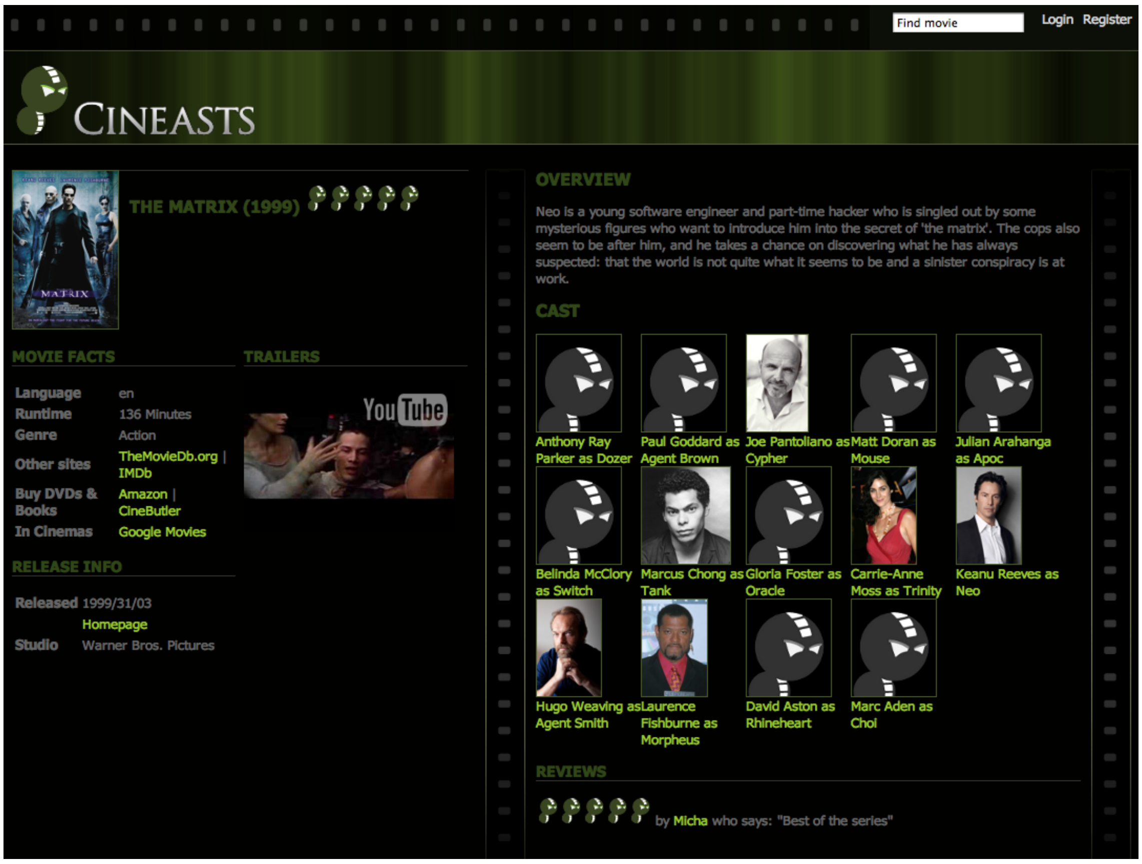
Showing off

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 11.1. 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.getMovie(movieId);
    model.addAttribute("id", movieId);
    if (movie != null) {
        model.addAttribute("movie", movie);
        model.addAttribute("stars", movie.getStars());
    }
    return "/movies/show";
}
```

The UI had now evolved to this:



THE MATRIX (1999)

MOVIE FACTS

Language	en
Runtime	136 Minutes
Genre	Action
Other sites	TheMovieDb.org | IMDb
Buy DVDs & Books	Amazon | CineButler
In Cinemas	Google Movies

TRAILERS

RELEASE INFO

Released	1999/31/03
	Homepage
Studio	Warner Bros. Pictures

OVERVIEW

Neo is a young software engineer and part-time hacker who is singled out by some mysterious figures who want to introduce him into the secret of 'the matrix'. The cops also seem to be after him, and he takes a chance on discovering what he has always suspected: that the world is not quite what it seems to be and a sinister conspiracy is at work.

CAST

Anthony Ray Parker as Dozer
Paul Goddard as Agent Brown
Joe Pantoliano as Cypher
Matt Doran as Mouse
Julian Arahanga as Apoc
Belinda McClory as Switch
Marcus Chong as Tank
Gloria Foster as Oracle
Carrie-Anne Moss as Trinity
Keanu Reeves as Neo
Hugo Weaving as Agent Smith
Laurence Fishburne as Morpheus
David Aston as Rhineheart
Marc Aden as Choi

REVIEWS

by Micha who says: "Best of the series"

11.1. Searching

The next thing was to allow users to search for movies, so we needed some fulltext search capabilities. As the 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(fulltext = true). We got an exception back telling us that we have to specify a separate index name. So we simply changed it to @Indexed(fulltext = true, indexName = "search"). The corresponding repository method is called findAllByQuery. To restrict the size of the returned set we simply added a limit that truncates the result.

Example 11.2. Searching for movies

```
public class CineastRepository {
    ....
    public void List<Movie> findMovies(String query, int count) {
        List<Movie> movies=new ArrayList<Movie>(count);
        ClosableIterable<Movie> searchResults = movieRepository.findAllByQuery("title", query);
        for (Movie movie : searchResults) {
            movies.add(movie);
            if (count-- == 0) break;
        }
        searchResults.close();
        return movies;
    }
}
```

11.2. Listing results

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

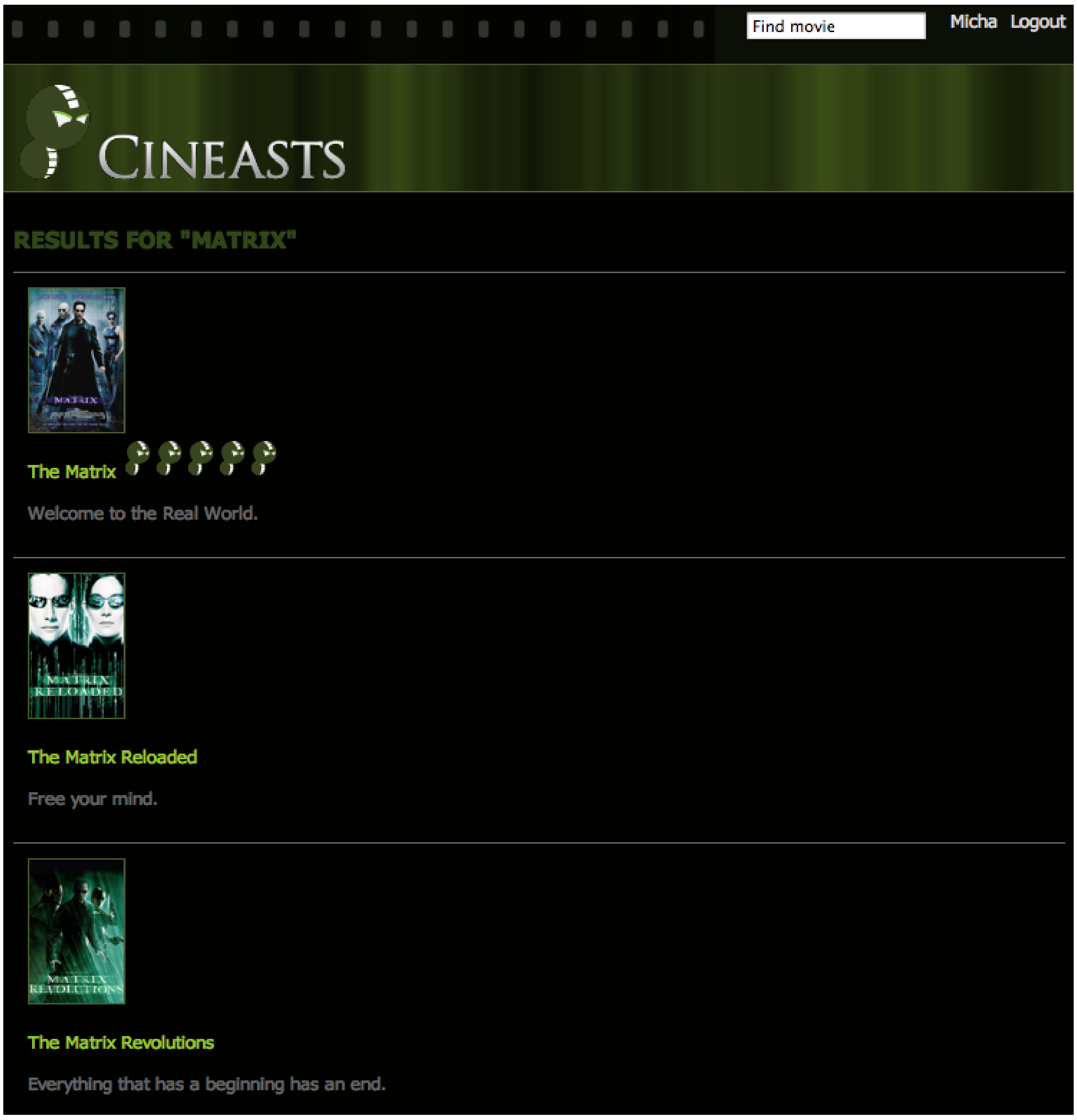
Example 11.3. Search controller

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

Example 11.4. 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="${movie.description}" escapeXml="true" />
            </dd>
        </c:forEach>
        </dl>
    </c:when>
    <c:otherwise>
        No movies found for query &quot;${query}&quot;.
    </c:otherwise>
</c:choose>
```

The UI now looked like this:



RESULTS FOR "MATRIX"

The Matrix

Welcome to the Real World.

The Matrix Reloaded

Free your mind.

The Matrix Revolutions

Everything that has a beginning has an end.

Chapter 12. Adding social

Movies 2.0

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

12.1. Users

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

Example 12.1. Social entities

```
@NodeEntity
class User {
    @Indexed String login;
    String name;
    String password;

    @RelatedToVia(elementClass = Rating.class, type = RATED)
    Iterable<Rating> ratings;

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

    public Rating rate(Movie movie, int stars, String comment) {
        return relateTo(movie, Rating.class, "RATED").rate(stars, comment);
    }
    public void befriend(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 12.2. Populate users and ratings

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

    User me = new User("micha", "Micha", "password",
        User.Roles.ROLE_ADMIN, User.Roles.ROLE_USER).persist();
    Rating awesome = me.rate(forestGump, 5, "Awesome");

    User ollie = new User("ollie", "Olliver", "password", User.Roles.ROLE_USER).persist();
    ollie.rate(forestGump, 2, "ok");
    me.addFriend(ollie);
    return asList(forestGump);
}
```

12.2. 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 12.3. Getting the rating of a movie

```
class Movie {
    ...

    @RelatedToVia(elementClass=Rating.class, type="RATED", direction = Direction.INCOMING)
    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 13. Adding Security

Protecting assets

To have a 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 that used a repository for 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 13.1. 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 13.2. 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>
```

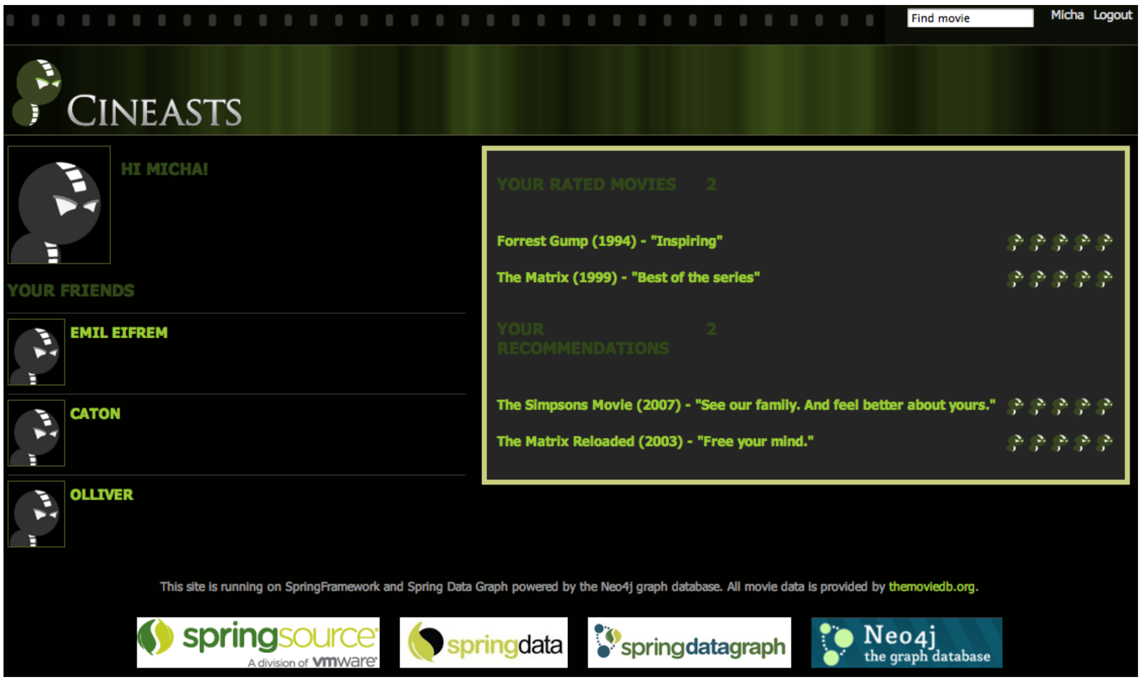
Example 13.3. Spring Security applicationContext-security.xml

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

<security:http auto-config="true" access-denied-page="/auth/denied"> <!-- use-expressions="true" -->
    <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=
    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="userDetailsService">
        <security:password-encoder hash="md5">
            <security:salt-source system-wide="cewuiqwzie"/>
        </security:password-encoder>
    </security:authentication-provider>
</security:authentication-manager>

<bean id="userDetailsService" class="org.neo4j.movies.service.CineastsUserDetailsService"/>
```

Example 13.4. UserDetailsService and UserDetails implementation

```
@Service
public class CineastsUserDetailsService implements UserDetailsService, InitializingBean {

    @Autowired private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException, DataAccessEx
        final User user = findUser(login);
        if (user==null) throw new UsernameNotFoundException("Username not found",login);
        return new CineastsUserDetails(user);
    }

    public User findUser(String login) {
        return userRepository.findByPropertyValue("login",login);
    }
    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:



YOUR RATED MOVIES 2

Forrest Gump (1994) - "Inspiring"

The Matrix (1999) - "Best of the series"

YOUR RECOMMENDATIONS 2

The Simpsons Movie (2007) - "See our family. And feel better about yours."

The Matrix Reloaded (2003) - "Free your mind."

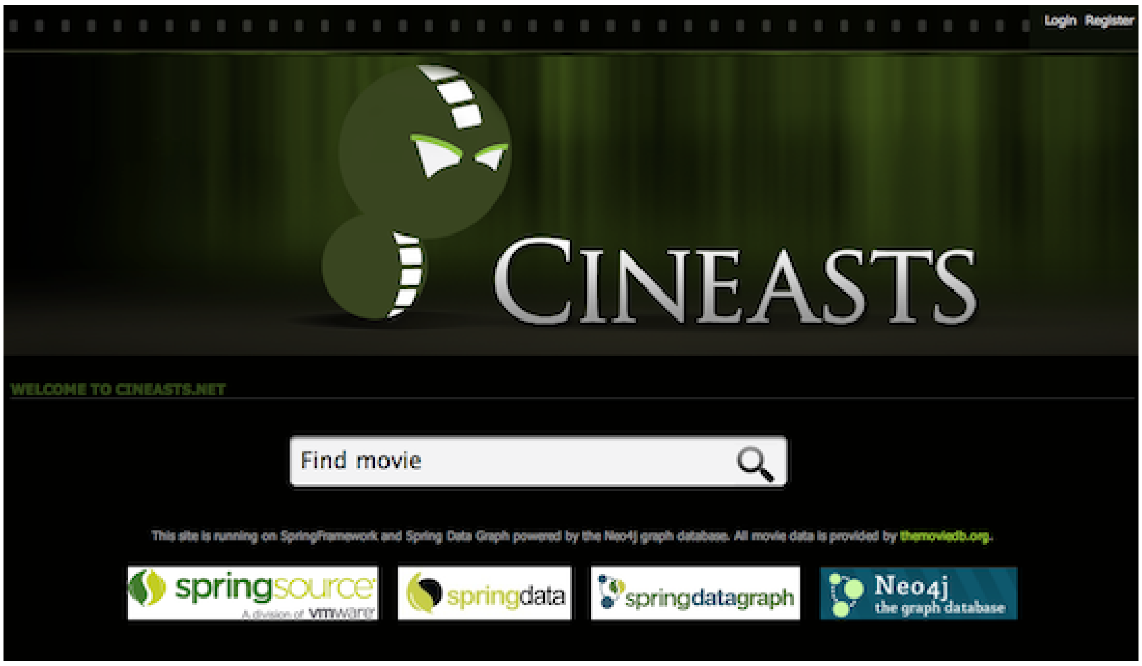
This site is running on SpringFramework and Spring Data Graph powered by the Neo4j graph database. All movie data is provided by themoviedb.org.

Chapter 14. More UI

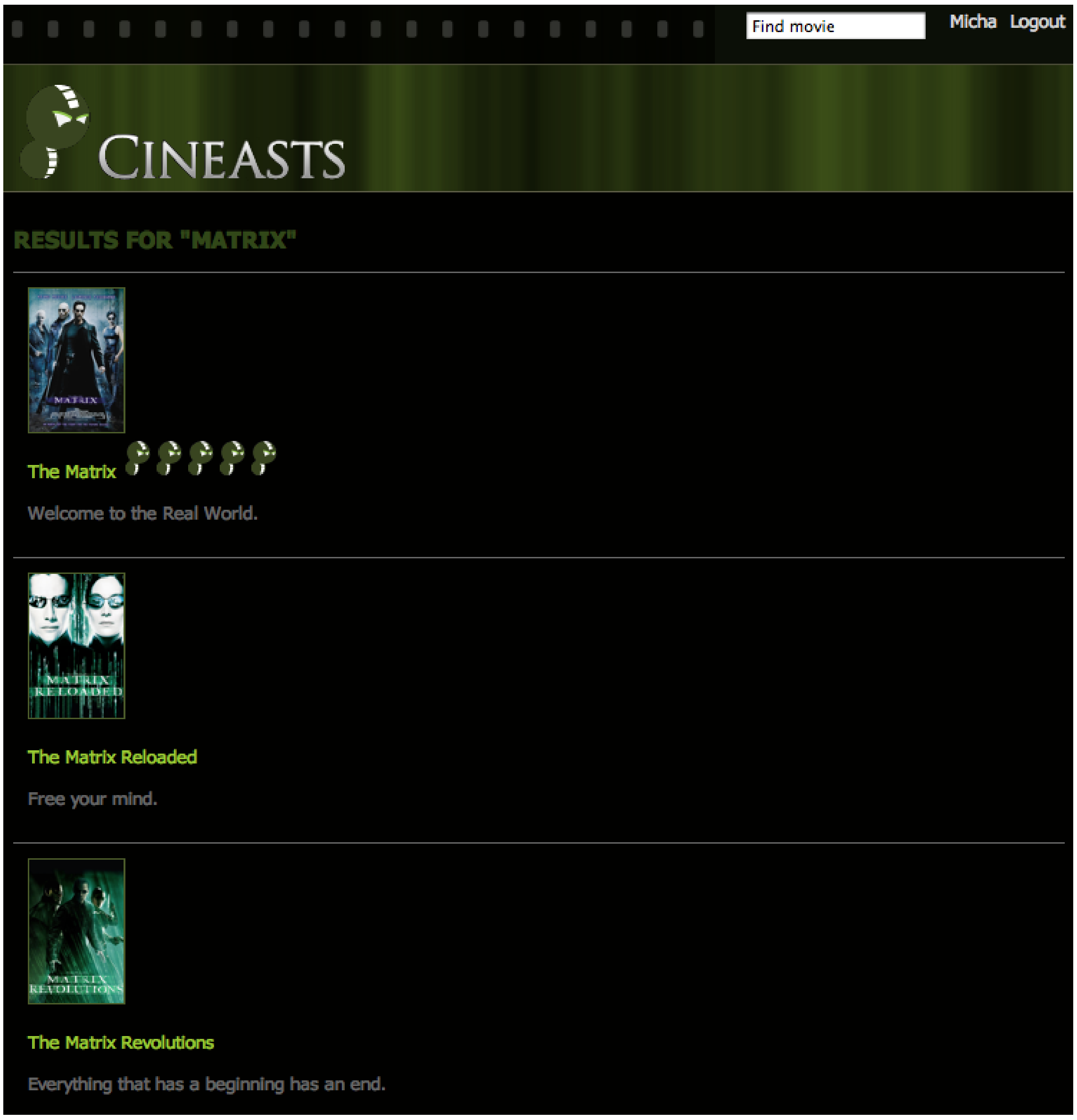
Oh the glamour

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.

Some noteworthy things. Since Spring Data Graph reads through down to the database for property and relationship access, we tried to minimize that by using `<c:var/>` several times. The app contains very little javascript / ajax code right now, that will change when it moves ahead.



This site is running on SpringFramework and Spring Data Graph powered by the Neo4j graph database. All movie data is provided by themoviedb.org.



RESULTS FOR "MATRIX"

The Matrix

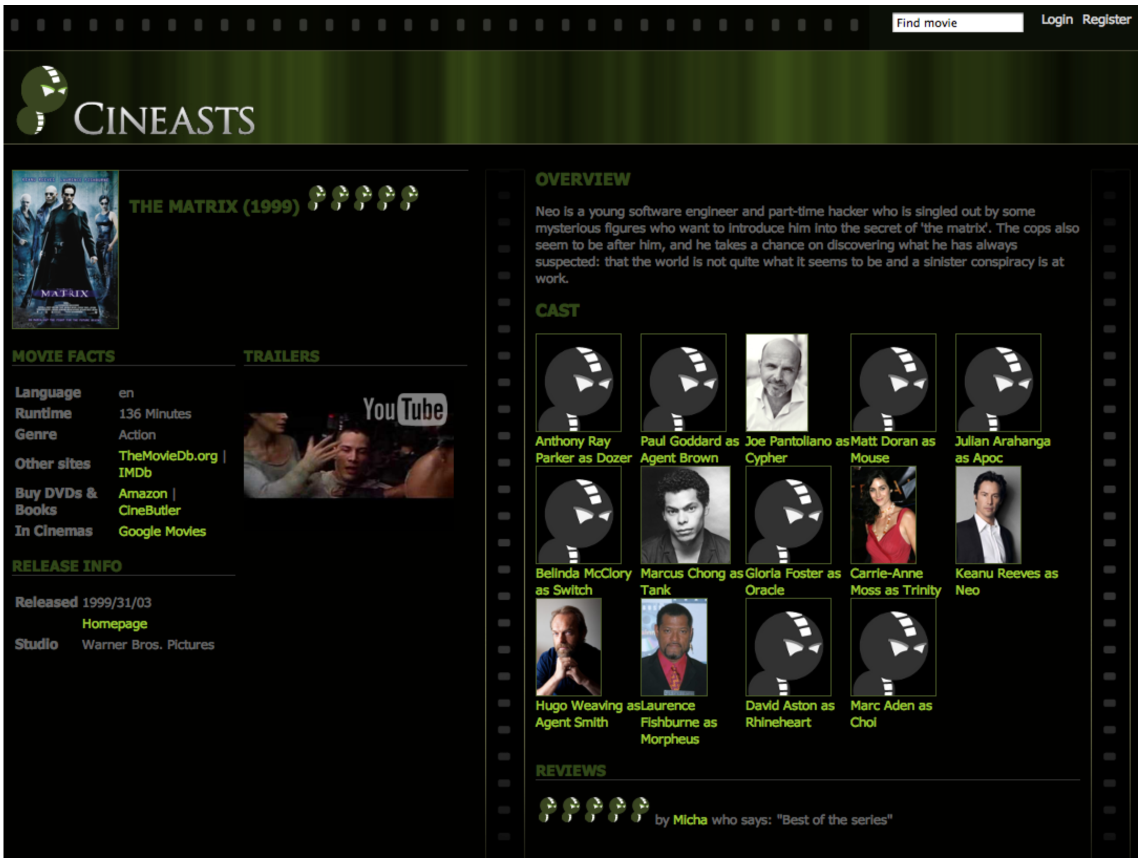
Welcome to the Real World.

The Matrix Reloaded

Free your mind.

The Matrix Revolutions

Everything that has a beginning has an end.



Find movie Login Register

CINEASTS

THE MATRIX (1999)

MOVIE FACTS

Language	en
Runtime	136 Minutes
Genre	Action
Other sites	TheMovieDb.org | IMDb
Buy DVDs & Books	Amazon | CineButler
In Cinemas	Google Movies

TRAILERS

RELEASE INFO

Released	1999/31/03
	Homepage
Studio	Warner Bros. Pictures

OVERVIEW

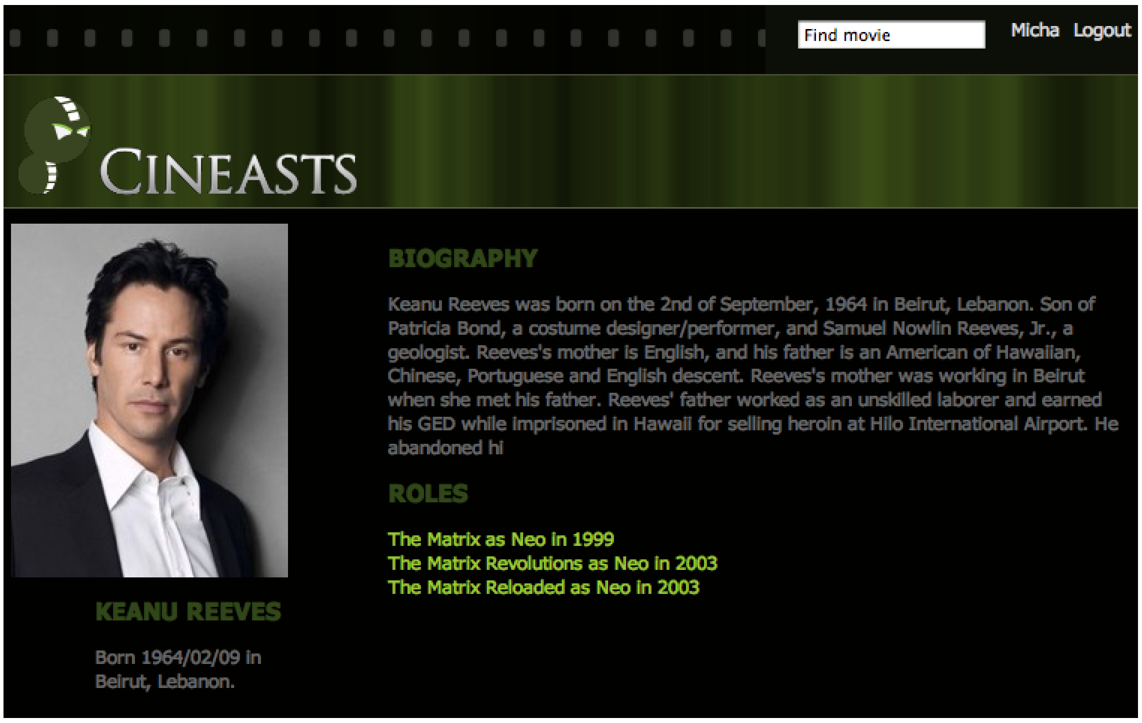
Neo is a young software engineer and part-time hacker who is singled out by some mysterious figures who want to introduce him into the secret of 'the matrix'. The cops also seem to be after him, and he takes a chance on discovering what he has always suspected: that the world is not quite what it seems to be and a sinister conspiracy is at work.

CAST

Anthony Ray Parker as Dozer
Paul Goddard as Agent Brown
Joe Pantoliano as Cypher
Matt Doran as Mouse
Julian Arahanga as Apoc
Belinda McClory as Switch
Marcus Chong as Tank
Gloria Foster as Oracle
Carrie-Anne Moss as Trinity
Keanu Reeves as Neo
Hugo Weaving as Agent Smith
Laurence Fishburne as Morpheus
David Aston as Rhineheart
Marc Aden as Choi

REVIEWS

by Micha who says: "Best of the series"



Find movie Micha Logout

CINEASTS

KEANU REEVES

Born 1964/02/09 in Beirut, Lebanon.

BIOGRAPHY

Keanu Reeves was born on the 2nd of September, 1964 in Beirut, Lebanon. Son of Patricia Bond, a costume designer/performer, and Samuel Nowlin Reeves, Jr., a geologist. Reeves's mother is English, and his father is an American of Hawaiian, Chinese, Portuguese and English descent. Reeves's mother was working in Beirut when she met his father. Reeves' father worked as an unskilled laborer and earned his GED while imprisoned in Hawaii for selling heroin at Hilo International Airport. He abandoned hi

ROLES

The Matrix as Neo in 1999
The Matrix Revolutions as Neo in 2003
The Matrix Reloaded as Neo in 2003

Chapter 15. 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 15.1. 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 lo
"keywords":["terror", "lebende leichen", "obsession", "camcorder", "firemen", "reality tv ", "bite", "
"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":"4cc8df41
....
"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/390/4c0157fa017a3c702d001390/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 15.2. 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/4bc9185d017a3c57fe0094ca/bud-abbott-lou-costello-meet-f
"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 15.3. Importing the data

```
@Transactional
public Movie importMovie(String movieId) {
    Movie movie = repository.getMovie(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 foun
    movieDbJsonMapper.mapToMovie(data, movie);
    movie.persist();
    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");
        Roles job = entry.get("job");
        Person person = importPerson(id);
        switch (job) {
            case DIRECTED:
                person.directed(movie);
                break;
            case ACTS_IN:
                person.playedIn(movie, (String) entry.get("character"));
                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 16. 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 friends liked (and their friends too, but with less importance). The second recommendation was for new friends that also liked the movies that we liked most.

Doing these kinds of ranking algorithms is a lot of fun with graph databases. The algorithms are implemented by traversing the graph in a certain order, collecting information on the go, and deciding which paths to follow and what to include in the results.

We were only interested in recommendations of a certain degree of friends.

Example 16.1. Recommendations

```
public Map<Movie,Integer> recommendMovies(User user, final int ratingDistance) {
    final DynamicRelationshipType RATED = withName(User.RATED);
    final Map<Long,int[]> ratings=new HashMap<Long, int[]>();
    TraversalDescription traversal= Traversal.description().breadthFirst()
        .relationships(withName(User.FRIEND)).relationships(RATED, OUTGOING).evaluator(new Evaluator()

      public Evaluation evaluate(Path path) {
          final int length = path.length() - 1;
          if (length > ratingDistance) return Evaluation.EXCLUDE_AND_PRUNE; // only as far as requeste
          Relationship rating = path.lastRelationship();
          if (rating != null && rating.getType().equals(RATED)) { // process RATED relationships, not
              if (length == 0) return Evaluation.EXCLUDE_AND_PRUNE; // my rated movies
              final long movieId = rating.getEndNode().getId();
              int[] stars = ratings.get(movieId);
              if (stars == null) {
                  stars = new int[2];
                  ratings.put(movieId, stars);
              }
              int weight = ratingDistance - length; // aggregate for averaging, inverse to distance
              stars[0] += weight * (Integer) rating.getProperty("stars", 0);
              stars[1] += weight;
              return Evaluation.INCLUDE_AND_PRUNE;
          }
          return Evaluation.EXCLUDE_AND_CONTINUE;
      }
    });

    Map<Movie,Integer> result=new HashMap<Movie, Integer>();
    final Iterable<Movie> movies = movieRepository.findAllByTraversal(user, traversal); // lazy traver
    for (Movie movie : movies) { // assign movie to averaged rating
      final int[] stars = ratings.get(movie.getNodeId());
      result.put(movie, stars[0]/stars[1]);
    }
    return result;
}
```

The UserController simply called this method, added its results to the model, and the view rendered the recommendation alongside the user's own ratings.

Part II. Reference

This is the reference part of the book. It has information about the programming model, APIs, concepts, and annotations of Spring Data Graph.

Preface

The Spring Data Graph project, as part of the Spring Data initiative, aims to simplify development with graph databases. Like JPA, it uses annotations on simple POJO beans. The annotations activate the AspectJ aspects in the Spring Data Graph framework, mapping the POJO entities and their fields to nodes, relationships, and properties in the graph database.

Spring Data Graph allows anytime to drop down to the Neo4j-API level to execute functionality with the highest performance possible. For Integration of Neo4j and Grails/GORM please refer to the Neo4j grails plugin.

To get started with a simple application, only the basic annotations (see Section 19.2, “Defining node entities”) and the additional aspect-introduced entity methods (see Section 19.9, “Introduced methods”) are required. Basic knowledge of graph stores is needed to access advanced functionality like traversals.

Note

As Spring Data Graph is based on AspectJ and uses some advanced features of that toolset, please be aware of that. Please see the section on AspectJ (Section 19.1, “AspectJ support”) for details if you run into any problems.

Chapter 17. About Spring Data

Spring Data is a SpringSource project that aims to provide Spring's convenient programming model and well known conventions for NOSQL databases. Currently there is support for graph (e.g. Neo4j), key-value (e.g. Redis), document (e.g. MongoDB) and relational (e.g. Oracle) databases. Mark Pollack, the author of Spring.NET, is the project lead for the Spring Data project.

Chapter 18. Introduction to Neo4j

18.1. 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 domain models, 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 one 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.

18.2. About Neo4j

Neo4j is a 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 brain, 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 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 for traversing in the node space.
- can be deployed as a standalone server or an embedded database with a very small distribution footprint (~700k jar).
- has a 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.

18.3. GraphDatabaseService

The interface `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.

18.4. Creating nodes and relationships

Using the API of GraphDatabaseService, it is easy to create nodes and relate them to each other. Relationships are typed. Both nodes and relationships can have properties. Property values can be primitive Java types and Strings, or arrays of Java primitives or Strings. Node creation and modification has to happen within a transaction, while reading from the graph store can be done with or without a transaction.

Example 18.1. Neo4j usage

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

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

18.5. Graph traversal

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

Example 18.2. 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);
}
```

18.6. Indexing

The best way for retrieving start nodes for traversals is by using Neo4j's integrated index facilities. The GraphDatabaseService provides access to the 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 Graph provides automatic indexing via the @Indexed annotation, eliminating the need for manual index management.

Note

Modifying Neo4j indexes also requires transactions.

Example 18.3. Index usage

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

Chapter 19. Programming model

This chapter covers the fundamentals of the programming model behind Spring Data Graph. It discusses the AspectJ features used and the annotations provided by Spring Data Graph and how to use them. Examples for this section are taken from the imdb project of Spring Data Graph examples.

19.1. AspectJ support

Behind the scenes, Spring Data Graph leverages AspectJ aspects to modify the behavior of simple annotated POJO entities (see Chapter 25, *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 Graph can retrieve the information from the entity's backing node or relationship in the database.

The aspect introduces some internal fields and some public methods (see Section 19.9, “Introduced methods”) in the entities, such as `entity.getPersistentState()` and `entity.relateTo`. It also introduces repository methods like `find(Class<? extends NodeEntity>, TraversalDescription)`, and `equals()` and `hashCode` delegation, making `equals()` honor the backing state.

Spring Data Graph 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 code. 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 are various layers of caching involved as well, so it handles repeat instantiation efficiently.

19.1.1. AspectJ IDE support

As Spring Data Graph uses some advanced features of AspectJ, users may experience issues with their IDE reporting errors where there in fact 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.

IDE's not providing the full AJ support might mark parts of your code as errors. You should rely on your build-system and test 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: http://download.eclipse.org/tools/ajdt/36/update/ (it might be necessary to use the latest development snapshot of the plugin http://download.eclipse.org/tools/ajdt/36/dev/update). The current version that does not show incorrect errors is AspectJ 1.6.12.M1 (included in STS 2.7.0.M2), previous versions are reported to mislead the user.

The AspectJ support in IntelliJ IDEA lacks some of the features. JetBrains is working on improving the situation in their upcoming 10.5 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.

19.2. Defining node entities

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

19.2.1. @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 overridden 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 Chapter 21, *Cross-store persistence* for more information.

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

Example 19.1. Simple node entity

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

19.2.2. @GraphProperty: Optional annotation for property fields

It is not necessary to annotate data fields, as they are persisted by default; all fields that contain primitive values are persisted directly to the graph. All fields convertible to String using the Spring conversion services will be stored as a string. Spring Data Graph includes a custom conversion factory that comes with converters for `Enums` and `Dates`. Transient fields are not persisted.

Currently there is no support for handling arbitrary collections of primitive or convertable values. Support for this will be added by the 1.1. release.

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`.

19.2.3. @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 Section 19.4, “Indexing” and Section 19.5, “CRUD with repositories” for more information.

19.2.4. @GraphQuery: fields as query result views

The `@GraphQuery` annotation leverages the delegation infrastructure used by the Spring Data Graph aspects. It provides dynamic fields which, when accessed, return the values selected by the provided query language expression. The provided query must contain a placeholder for the id of the current entity `start n=(%d) match n-[:FRIEND]->friend return friend` As graph queries can return variable number of entities the annotation can be put onto fields with a single value, an Iterable of a type or an Iterable of `Map<String,Object>`. The class of the resulting node entities must right now provided with the `elementClass` attribute. Additional parameters are added to the query with Java's String.format substitution.

Example 19.2. @GraphQuery from a node entity

```
@NodeEntity
public class Group {
    @GraphQuery(value = "start n=(%d) match (n)-[:%s]->(friend) return friend",
            elementClass = Person.class, params = "FRIEND")
    private Iterable<Person> friends;
}
```

19.2.5. @GraphTraversal: fields as traversal result views

The `@GraphTraversal` annotation leverages the delegation infrastructure used by the Spring Data Graph aspects. It provides dynamic fields which, when accessed, return an Iterable of node 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 `traversalBuilder` attribute. The class of the resulting node entities must be provided with the `elementClass` attribute.

Example 19.3. @GraphTraversal from a node entity

```
@NodeEntity
public class Group {
    @GraphTraversal(traversalBuilder = 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());
        }
    }
}
```

19.3. 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 Graph 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 Graph reflects this accordingly.

19.3.1. @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 Graph 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. If the field is set to `null`, the relationship is removed.

Example 19.4. Single relationship field

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

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 `java.util.Set<T>`, and read-only fields are `java.lang.Iterable<T>`, where T is a @NodeEntity-annotated class. The Java implementation of generics uses type erasure, meaning that the type parameters are typically not available at runtime. Therefore, the `elementClass` attribute must be specified on the annotation, which must always be present for 1:N fields.

Example 19.5. Node entity with relationships

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

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

Fields referencing other entities should not be manually initialized, as they are managed by Spring Data Graph under the hood. 1:N fields can be accessed immediately, and Spring Data Graph will provide a java.util.Set representing the relationships. If the returned set is modified, the changes are reflected in the graph. Spring Data Graph also ensures that there is only one relationship of a given type between any two given entities.

Note

Before an entity has been attached with `persist()` for the first time, it will not have its state managed by Spring Data Graph. 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 you use an Interface as target type for the `Set` and/or as `elementClass` please make sure that it implements `NodeBacked` either by extending that Super-Interface manually or by annotating the Interface with `@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.

The relationships can also be accessed by using the aspect-introduced methods `entity.getRelationshipTo(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.

Note

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

19.3.2. @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 link node entities together via relationships, but it provides no way of accessing the relationships themselves.

Relationship entities cannot be instantiated directly but are rather created via node entities, either by @RelatedToVia-annotated fields (see Section 19.3.3, “@RelatedToVia: Accessing relationship entities”), or by the introduced `entity.relateTo(target, relationshipClass, type)` and `entity.getRelationshipTo(target, relationshipClass, type)` methods (see Section 19.9, “Introduced methods”).

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.

Example 19.6. Relationship entity

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

@RelationshipEntity
public class Role {
    String title;

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

19.3.3. @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 `java.lang.Iterable<T>`, where T is a `@RelationshipEntity`-annotated class. These fields provide read-only access to relationship entities.

Example 19.7. Accessing relationship entities using @RelatedToVia

```
@NodeEntity
public class Actor {
    @RelatedToVia(type = "ACTS_IN", elementClass = Role.class)
    private Iterable<Role> roles;

    public Role playedIn(Movie movie, String title) {
        Role role = relateTo(movie, Role.class, "ACTS_IN");
        role.setTitle(title);
        return role;
    }
}
```

19.4. Indexing

The Neo4j graph database can use different so-called index providers for exact lookups and fulltext searches. Lucene is the default index provider implementation. Each named index is configured to be fulltext or exact.

19.4.1. Exact and numeric index

When using the standard Neo4j API, nodes and relationships have to be manually indexed with key-value pairs, typically being the property name and value. When using Spring Data Graph, 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 index every time an indexed field changes.

Numerical fields are indexed numerically so that they are available for range queries. All other fields are indexed with their string representation.

The @Indexed annotation also provides the option of using a custom index. 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.

The indexes can be queried by using a repository (see Section 19.5, “CRUD with repositories”). Typically, the repository is an instance of `org.springframework.data.graph.neo4j.repository.DirectGraphRepositoryFactory`. 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).

Example 19.8. Indexing entities

```
@NodeEntity
class Person {
    @Indexed(indexName = "people") String name;
    @Indexed int age;
}

GraphRepository<Person> graphRepository = graphRepositoryFactory
        .createGraphRepository(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);
}
```

19.4.2. Fulltext indexes

Spring Data Graph also supports fulltext indexes. By default, indexed fields are stored in an exact lookup index. To have them analyzed and prepared for fulltext search, the `@Indexed` annotation has the boolean `fulltext` attribute. 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 19.9. Fulltext indexing

```
@NodeEntity
class Person {
    @Indexed(indexName = "person-name", fulltext=true) String name;
}

GraphRepository<Person> graphRepository = graphRepositoryFactory
        .createGraphRepository(Person.class);

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

Note

Please note that indexes are currently created on demand, so whenever an index that doesn't exist is requested from a query or get operation it is created. This is subject to change but has currently the implication that those indexes won't be configured as fulltext which causes subsequent fulltext updates to those indexes to fail.

19.4.3. Manual index access

The index for a domain class is also available from `GraphDatabaseContext` 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.

Example 19.10. Manual index usage

```
@Autowired GraphDatabaseContext gdc;

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

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

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

19.4.4. Indexing in Neo4jTemplate

Neo4jTemplate also offers index support, providing auto-indexing for fields at creation time. There is an `autoIndex` method that can also add indexes for a set of fields in one go.

For querying the index, the template offers query methods that take either the exact match parameters or a query object/expression, and push the results wrapped uniformly as Paths to the supplied `PathMapper` to be converted or collected.

19.5. CRUD with repositories

The repositories provided by Spring Data Graph 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 Graph comes with typed repository implementations that provide methods for locating node and relationship entities. There are 3 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.

`GraphRepository` is a convenience repository interface, extending `CRUDRepository`, `IndexRepository`, and `TraversalRepository`. Generally, it has all the desired repository methods. If named index operations are required, then `NamedIndexRepository` may also be included.

19.5.1. CRUDRepository

`CRUDRepository` delegates to the configured `TypeRepresentationStrategy` (see Section 19.8, “Entity type representation”) for type based queries.

Load an instance via a Neo4j node id
: `T findOne(id)`

Check for existence of a Neo4j node id
: `boolean exists(id)`

Iterate over all nodes of a node entity type
: `Iterable<T> findAll()` (supported in future versions: `Iterable<T> findAll(Sort)` and `Page<T> findAll(Pageable)`)

Count the instances of a node entity type
: `Long count()`

Save a graph entity
: `T save(T)` and `Iterable<T> save(Iterable<T>)`

Delete a graph entity
: `void delete(T)`, `void; delete(Iterable<T>)`, and `deleteAll()`

Important to note here is that the `save`, `delete`, and `deleteAll` methods are only there to conform to the `org.springframework.data.repository.Repository` interface. The recommended way of saving and deleting entities is by using `entity.persist()` and `entity.remove()`.

19.5.2. 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
: `Iterable<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)
: `Iterable<T> findAllByRange(key, from, to)`

Iterate over all indexed entity instances with field values matching the given fulltext string or QueryContext query
: `Iterable<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.

19.5.3. TraversalRepository

`TraversalRepository` delegates to the Neo4j traversal framework.

Iterate over a traversal result
: `Iterable<T> findAllByTraversal(startEntity, traversalDescription)`

19.5.4. Creating repositories

The `Repository` instances are either created manually via a `DirectGraphRepositoryFactory`, bound to a concrete node or relationship entity class. The `DirectGraphRepositoryFactory` is configured in the Spring context and can be injected.

Example 19.11. Using GraphRepositories

```
GraphRepository<Person> graphRepository = graphRepositoryFactory
        .createGraphRepository(Person.class);

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

Person dave = graphRepository.findOne(123);

Long numberOfPeople = graphRepository.count();

Person mark = graphRepository.findByPropertyValue("name", "mark");

Iterable<Person> devs = graphRepository.findAllByProperyValue("occupation", "developer");

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

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

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

19.5.5. 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 19.12. Composing repositories

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

// 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
// (graphDatabaseContext reference is optional)
<datagraph:repositories base-package="org.springframework.data.graph.neo4j"
    graph-database-context-ref="graphDatabaseContext"/>

// have it injected
@Autowired
PersonRepository personRepository;

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

Person dave=personRepository.findOne(123);

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

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

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

19.6. Transactions

Neo4j is a transactional database, only allowing modifications to be performed within transaction boundaries. Reading data does however not require transactions.

Spring Data Graph 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

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 Graph.

Example 19.13. Simple transaction manager configuration

```
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager">
        <bean class="org.neo4j.kernel.impl.transaction.SpringTransactionManager">
            <constructor-arg ref="graphDatabaseService"/>
        </bean>
    </property>
    <property name="userTransaction">
        <bean class="org.neo4j.kernel.impl.transaction.UserTransactionImpl">
            <constructor-arg ref="graphDatabaseService"/>
        </bean>
    </property>
</bean>

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

For scenarios with multiple transactional resources there are two options. The first option is to have Neo4j participate in the externally configured transaction manager by 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 19.14. 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.graph.neo4j.transaction.JotmFactoryBean"/>
    </property>
</bean>

<bean 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 Graph. 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 19.15. ChainedTransactionManager example

```
<bean id="jpaTransactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<bean id="jtaTransactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager">
        <bean class="org.neo4j.kernel.impl.transaction.SpringTransactionManager">
            <constructor-arg ref="graphDatabaseService" />
        </bean>
    </property>
    <property name="userTransaction">
        <bean  class="org.neo4j.kernel.impl.transaction.UserTransactionImpl">
            <constructor-arg ref="graphDatabaseService" />
        </bean>
    </property>
</bean>
<bean id="transactionManager"
        class="org.springframework.data.graph.neo4j.transaction.ChainedTransactionManager">
    <constructor-arg>
        <list>
            <ref bean="jpaTransactionManager"/>
            <ref bean="jtaTransactionManager"/>
        </list>
    </constructor-arg>
</bean>

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

19.7. Detached node entities

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

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 19.16. Persisting entities

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

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

19.7.1. 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 19.17. 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 Graph 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 persist 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 parametrizable in the future).

If the relationships form a cycle, then the entities will first 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 19.18. 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.

19.8. 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. Some of this type information is saved in the graph database.

Implementations of `TypeRepresentationStrategy` take care of persisting this information on entity instance creation. They also provide the repository methods that use this type information to perform their operations, like findAll and count.

There are three available implementations for node entities to choose from.

- `IndexingNodeTypeRepresentationStrategy`

  Stores entity types in the integrated index. Each entity node gets indexed with its type and any supertypes that are also`@NodeEntity`-annotated. The special index used for this is called`__types__`. Additionally, in order to get the type of an entity node, each node has a property `__type__` with the type of that entity.

- `SubReferenceNodeTypeRepresentationStrategy`

  Stores entity types in a tree in the graph representing the type 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, same behavior as the corresponding ones above:

- `IndexingRelationshipTypeRepresentationStrategy`

- `NoopRelationshipTypeRepresentationStrategy`

Spring Data Graph 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. 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.

19.9. Introduced methods

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 `EntityPaths` of the traversal result bound to the provided start and end-node-entity types
: `Iterable<EntityPath> findAllPathsByTraversal(traversalDescription)`

19.10. 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 benefitial.

Spring Data Graph 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 data changes are reflected immediately.

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 19.19. Projection of entities

```
@NodeEntity
class Trainee {
    String name;
    @RelatedTo(elementClass=Training.class);
    Set<Training> trainings;
}

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

19.11. Bean validation (JSR-303)

Spring Data Graph supports property-based validation support. When a property is changed, 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 `GraphDatabaseContext`.

Example 19.20. Bean validation

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

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

Chapter 20. Environment setup

Spring Data Graph dramatically simplifies development, but some setup is naturally required. For building the application, Maven needs to be configured to include the Spring Data Graph dependencies, and configure the AspectJ weaving. After the build setup is complete, the Spring application needs to be configured to make use of Spring Data Graph.

Spring Data Graph projects can be built using maven, we also added means to build them with gradle and ant/ivy.

20.1. Gradle configuration

The necessary build plugin to build Spring Data Graph projects with gradle is available as part of the SDG distribution or on github which makes the usage as easy as:

Example 20.1. Gradle Build Configuration

```
sourceCompatibility = 1.6
targetCompatibility = 1.6

springVersion = "3.0.5.RELEASE"
springDataGraphVersion = "1.1.0.M1"
aspectjVersion = "1.6.12.M1

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

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

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

20.2. Ant/Ivy configuration

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

Example 20.2. Ant/Ivy Build Configuration

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

<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.jar"/>
  </aspectpath>
 </iajc>
</target>
```

20.3. Maven configuration

Spring Data Graph projects are easiest to build with Apache Maven. The main dependencies are: Spring Data Graph itself, Spring Data Commons, parts of the Spring Framework, and the Neo4j graph database.

20.3.1. Repositories

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

Example 20.3. Spring milestone repository

```
<repository>
    <id>spring-maven-milestone</id>
    <name>Springframework Maven Repository</name>
    <url>http://maven.springframework.org/milestone</url>
</repository>
```

20.3.2. Dependencies

The dependency on `spring-data-neo4j` 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.

Example 20.4. Maven dependencies

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-neo4j</artifactId>
    <version>1.0.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.6.11.RELEASE</version>
</dependency>
```

20.3.3. AspectJ build configuration

Since Spring Data Graph uses AspectJ for build-time aspect weaving of entities, it is necessary to hook in the AspectJ Maven plugin to 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).

Example 20.5. AspectJ configuration

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.0</version>
    <dependencies>
        <!-- NB: You must use Maven 2.0.9 or above or these are ignored (see MNG-2972) -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.6.11.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>1.6.11.RELEASE</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-datastore-neo4j</artifactId>
            </aspectLibrary>
        </aspectLibraries>
        <source>1.6</source>
        <target>1.6</target>
    </configuration>
</plugin>
```

20.4. Spring configuration

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

20.4.1. XML namespace

The XML namespace can be used to configure Spring Data Graph. The `config` element provides an XML-based configuration of Spring Data Graph in one line. It has three attributes. `graphDatabaseService` points out the Neo4j instance to use. For convenience, `storeDirectory` can be set instead of `graphDatabaseService` to point to a directory where a new `EmbeddedGraphDatabase` will be created. For cross-store configuration, the `entityManagerFactory` attribute needs to be configured.

Example 20.6. 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:datagraph="http://www.springframework.org/schema/data/graph"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd
            http://www.springframework.org/schema/data/graph
            http://www.springframework.org/schema/data/graph/datagraph-1.0.xsd">

    <context:annotation-config/>
    <datagraph:config storeDirectory="target/config-test"/>

</beans>
```

Example 20.7. XML configuration with bean

```
<context:annotation-config/>

<bean id="graphDatabaseService" class="org.neo4j.kernel.EmbeddedGraphDatabase"
        destroy-method="shutdown">
    <constructor-arg index="0" value="target/config-test" />
</bean>

<datagraph:config graphDatabaseService="graphDatabaseService"/>
```

Example 20.8. 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>

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

20.4.2. Java-based bean configuration

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

Note

For those not familiar with Java-based bean metadata 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 Graph with Java-based bean metadata, 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`. The example below shows how to register the `@Configuration` `Neo4jConfiguration` class, as well as Spring's `ConfigurationClassPostProcessor` that transforms the `@Configuration` class to bean definitions.

Example 20.9. Java-based bean configuration

```
<beans ...>
    ...
    <tx:annotation-driven mode="aspectj" transaction-manager="transactionManager"/>
    <bean class="org.springframework.data.graph.neo4j.config.Neo4jConfiguration"/>

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

    <bean id="graphDatabaseService" class="org.neo4j.kernel.EmbeddedGraphDatabase"
          destroy-method="shutdown" scope="singleton">
        <constructor-arg index="0" value="target/config-test"/>
    </bean>
    ...
</beans>
```

Chapter 21. Cross-store persistence

The Spring Data Graph project support cross-store persistence, 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.

21.1. Partial entities

Partial graph persistence is achieved by restricting the Spring Data Graph 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 `persist()`.

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

21.2. Cross-store annotations

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

21.2.1. @NodeEntity(partial = "true")

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

21.2.2. @GraphProperty

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

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

Example 21.1. 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(type = "friends", elementClass = UserAccount.class)
    Set<UserAccount> friends;

    @RelatedToVia(type = "recommends", elementClass = Recommendation.class)
    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;

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

    @Transactional
    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;
    }
}
```

21.3. Configuring cross-store persistence

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

Example 21.2. Cross-store Spring configuration

```xml
<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/graph"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/data/graph
        http://www.springframework.org/schema/data/graph/datagraph-1.0.xsd
        ">

    <context:annotation-config/>

    <datagraph: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 22. Sample code

22.1. Introduction

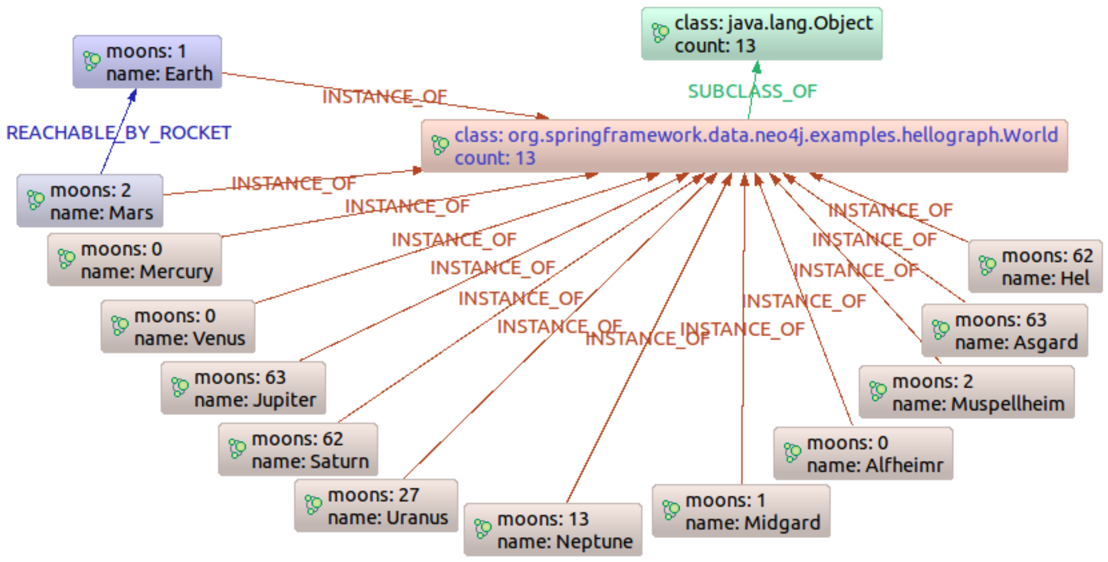
Spring Data Graph comes with a number of sample applications. The source code of the samples can be found on Github. The different sample projects are introduced below.

22.2. Hello Worlds sample application

The Hello Worlds sample application is a simple console application. It creates some worlds (node entities) and rocket routes (relationships) between worlds, all in a galaxy (the graph), and then prints them.

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

Executing the application creates the following graph in the graph database:

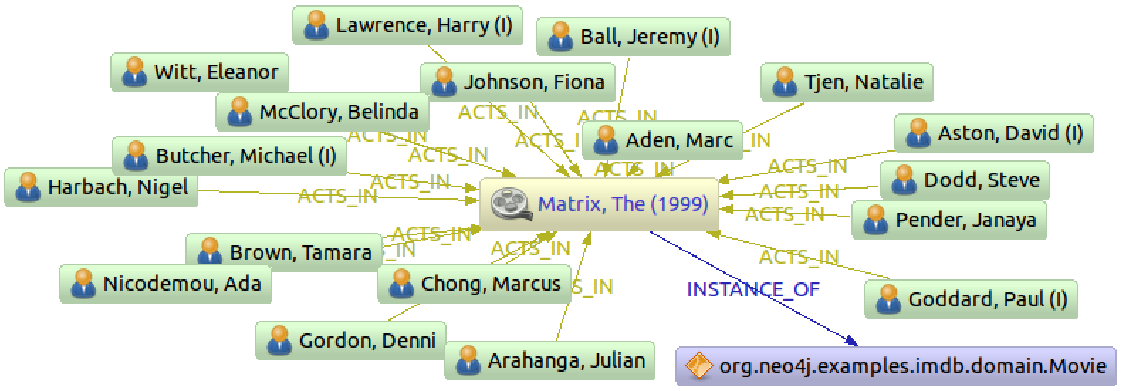


22.3. IMDB sample application

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 Bacon number of any given actor. This sample application shows the usage of Spring Data Graph in a more complex setting, using several annotated entities and relationships as well as indexes and graph traversals.

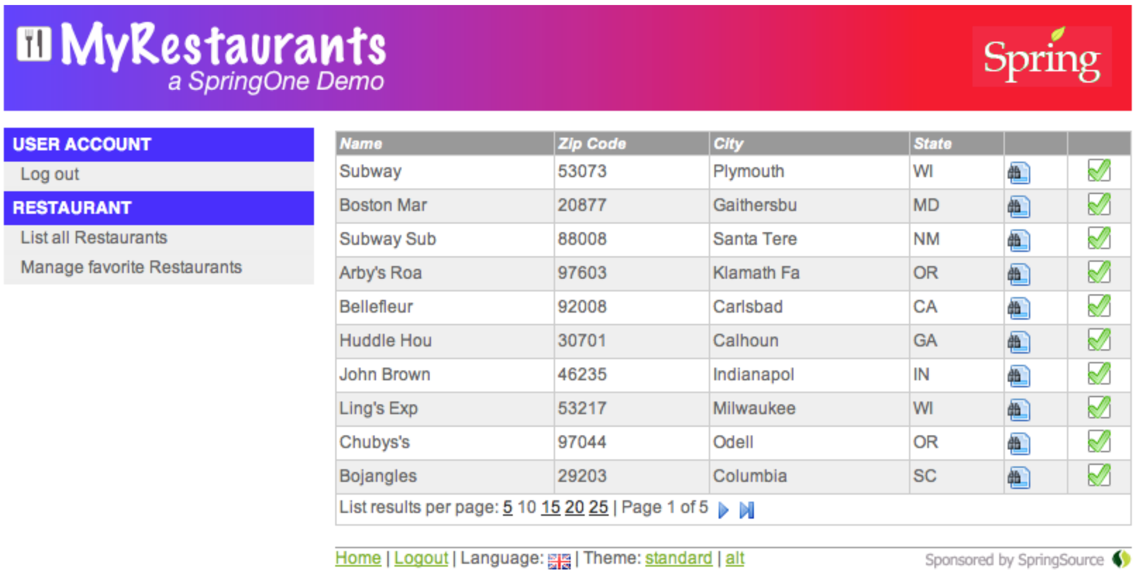
See the readme file for instructions on how to compile and run the application.

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



22.4. MyRestaurants sample application

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 (seeSection 22.5, “MyRestaurant-Social sample application”), and does therefore not use Spring Data Graph.



MyRestaurants
a SpringOne Demo
Spring

USER ACCOUNT
Log out
RESTAURANT
List all Restaurants
Manage favorite Restaurants

Name	Zip Code	City	State		
Subway	53073	Plymouth	WI		✓
Boston Mar	20877	Gaithersbu	MD		✓
Subway Sub	88008	Santa Tere	NM		✓
Arby's Roa	97603	Klamath Fa	OR		✓
Bellefleur	92008	Carlsbad	CA		✓
Huddle Hou	30701	Calhoun	GA		✓
John Brown	46235	Indianapol	IN		✓
Ling's Exp	53217	Milwaukee	WI		✓
Chubys's	97044	Odell	OR		✓
Bojangles	29203	Columbia	SC		✓

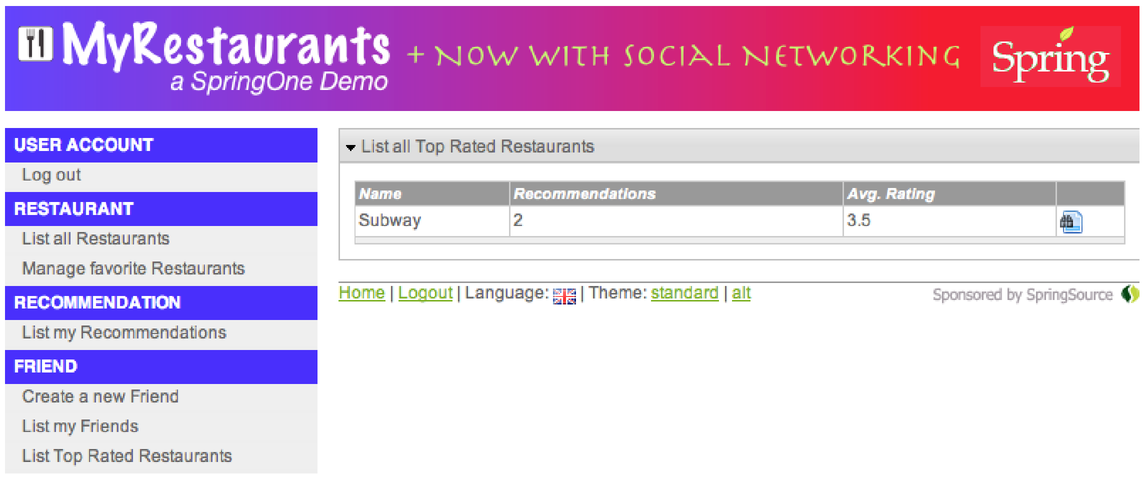
List results per page: 5 10 15 20 25 | Page 1 of 5

Home | Logout | Language: | Theme: standard | alt
Sponsored by SpringSource

22.5. MyRestaurant-Social sample application

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:



MyRestaurants + NOW WITH SOCIAL NETWORKING Spring
a SpringOne Demo

USER ACCOUNT
- Log out

RESTAURANT
- List all Restaurants
- Manage favorite Restaurants

RECOMMENDATION
- List my Recommendations

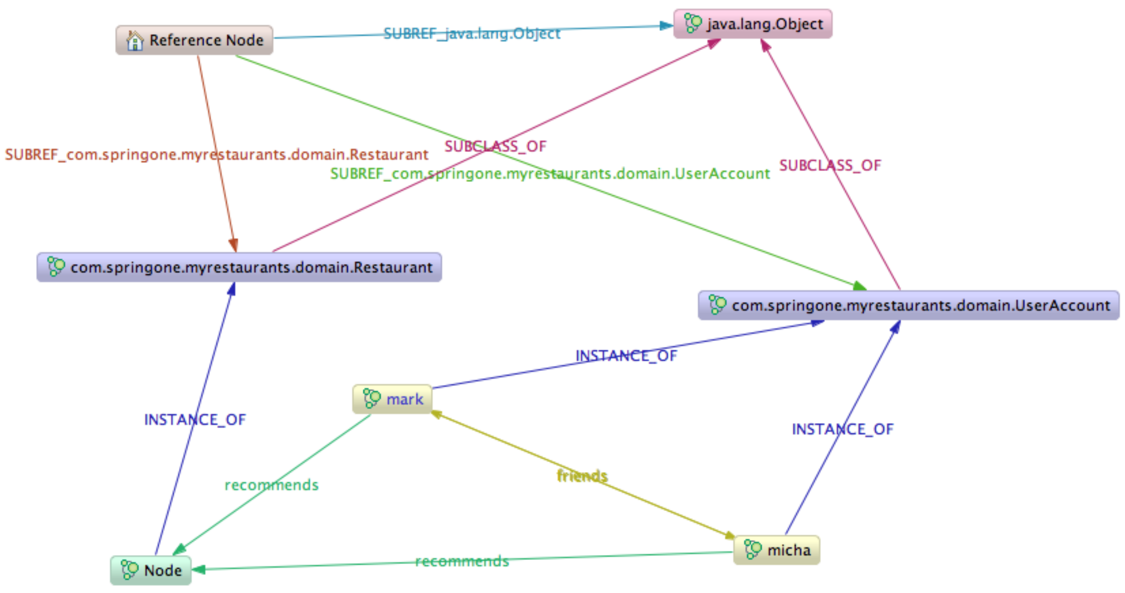
FRIEND
- Create a new Friend
- List my Friends
- List Top Rated Restaurants

List all Top Rated Restaurants

Name	Recommendations	Avg. Rating	
Subway	2	3.5	

Home | Logout | Language: | Theme: standard | alt

Sponsored by SpringSource


Reference Node
SUBREF_java.lang.Object
java.lang.Object
SUBREF_com.springone.myrestaurants.domain.Restaurant
SUBCLASS_OF
SUBREF_com.springone.myrestaurants.domain.UserAccount
SUBCLASS_OF
com.springone.myrestaurants.domain.Restaurant
com.springone.myrestaurants.domain.UserAccount
INSTANCE_OF
mark
INSTANCE_OF
INSTANCE_OF
recommends
friends
micha
Node
recommends

Chapter 23. 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 Graph instead of the Neo4j API directly.

23.1. When is Spring Data Graph right

The focus of Spring Data Graph 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.

Spring Data Graph was however not designed with a major focus on performance. It does add some overhead to pure graph operations. Something to keep in mind is 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).

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, so that only the first access gets a performance penalty.

Chapter 24. Neo4jTemplate

The `Neo4jTemplate` offers the convenient API of Spring templates for the Neo4j graph database.

24.1. 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 and optionally index certain fields.

Example 24.1. Neo4j template

```
import static org.springframework.data.graph.core.Property._;

Neo4jOperations neo = new Neo4jTemplate(graphDatabaseService);

Node michael = neo.createNode(_("name","Michael"));
Node mark = neo.createNode(_("name","Mark"));
Node thomas = neo.createNode(_("name","Thomas"));

neo.createRelationship(mark,thomas, WORKS_WITH, _("project","spring-data"));

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

assert "Mark".equals(neo.query("devs","name","Mark",new NodeNamePathMapper()));
```

24.2. Indexing

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

The `query()` 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 `query()` methods provide `Path` results to a PathMapper.

24.3. Graph traversal

The traversal methods are at the core of graph operations. As such, they are fully supported in the `Neo4jTemplate`. The `traverseNext()` method traverses to the direct neighbors of the start node, filtering the relationships according to the parameters.

The `traverse()` method covers the full traversal operation that takes a `TraversalDescription` (typically built with the `Traversal.description()` DSL) and runs it from the start node. Each path that is returned by the traversal is passed to the `PathMapper` to be converted into the desired type.

24.4. Path abstraction and PathMapper

For the querying operations Neo4jTemplate unifies the result with the `Path` abstraction that comes from Neo4j. Much like a result set, a path contains a chain of `nodes()` connected by `relationships()`, starting at a `startNode()` and ending at a `endNode()`. The `lastRelationship()` is also available separately. The `Path` abstraction also wraps results that contain just nodes or relationships.

Using implementations of `PathMapper<T>` and `PathMapper.WithoutResult` (comparable with `RowMapper` and `RowCallbackHandler`), the paths can be converted to arbitrary Java objects.

With `EntityPath` and `EntityMapper` there is also support for using node entities within the `Path` and `PathMapper` constructs.

24.5. Transactions

The `Neo4jTemplate` provides configurable implicit transactions for all its methods. By default it creates a transaction for each call (which is a no-op if there is already a transaction running). If you call the constructor with the `useExplicitTransactions` parameter set to true, it won't create any transactions so you have to provide them using `@Transactional` or the `TransactionTemplate`.

Chapter 25. AspectJ details

The object graph mapper of Spring Data Graph 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 Graph will not have to deal with that. Users don't have care about to 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. 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, 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 Graph 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 Section 19.9, “Introduced methods” for more information on the methods introduced.

Spring Data Graph 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 Graph.

Chapter 26. 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 REST API. Developers can integrate Spring Data Graph into the Neo4j server infrastructure in two ways: in an unmanaged server extension, or via the REST API.

26.1. Server Extension

When should you write a server extension? The default REST API is essentially a REST'ified representation of the Neo4j core API. It is nice for getting started, and for simpler scenarios. For more involved solutions that require high-volume access or more complex operations, writing a server extension that is able to process external parameters, do all the computations locally in the plugin, and then return just the relevant information to the calling client is preferable.

The Neo4j Server has two built-in extension mechanisms. It is possible to extend existing URI endpoints like the graph database, nodes, or relationships, adding new URIs or methods to those. This is achieved by writing a server plugin. This plugin type has some restrictions though.

For complete freedom in the implementation, an unmanaged extension can be used. Unmanaged extensions are essentially Jersey resource implementations. The resource constructors or methods can get the `GraphDatabaseService` injected to execute the necessary operations 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.

Example 26.1. Configuring an unmanaged extension

```
org.neo4j.server.thirdparty_jaxrs_classes=com.example.mypackage=/my-context
```

Running Spring Data Graph on the Neo4j Server is easy. You need to tell the server where to find the Spring context configuration file, and which beans from it to expose:

Example 26.2. Server plugin initialization

```
public class HelloWorldInitializer extends SpringPluginInitializer {
    public HelloWorldInitializer() {
        super(new String[]{"spring/helloWorldServer-Context.xml"},
              Pair.of("worldRepository", WorldRepository.class),
              Pair.of("graphRepositoryFactory", GraphRepositoryFactory.class));
    }
}
```

Now, your resources can be annotated with the beans they need, like this:

Example 26.3. Jersey resource

```
@Path( "/path" )
@POST
@Produces( MediaType.APPLICATION_JSON )
public void foo( @Context WorldRepository repo ) {
    ...
}
```

The `SpringPluginInitializer` merges the GraphDatabaseService with the Spring configuration and registers the named beans as Jersey `Injectables`. It is still necessary to list the initializer's fully qualified class name in a file named `META-INF/services/org.neo4j.server.plugins.PluginLifecycle`. The Neo4j Server can then pick up and run the initialization classes before the extensions are loaded.

26.2. Using Spring Data Graph as a REST client

Spring Data Graph can use a set of Java REST bindings which come as a drop in replacement for the GraphDatabaseService API. By simply configuring the `graphDatabaseService` to be a `RestGraphDatabase` pointing to a Neo4j Server instance.

Note

The Neo4j Server REST API does not allow for transactions to span across requests, which means that Spring Data Graph is not transactional when running with a `RestGraphDatabase`.

Please also keep in mind that performing graph operations via the REST-API is about one order of magnitude slower than location operations. Try to use the Neo4j-Query-Language or server-side traversals whenever possible (`RestTraversal`) for retrieving large sets of data. Future versions of Spring Data Graph will use the more performant batching as well as a binary protocol.

To set up your project to use the REST bindings, add this dependency to your pom.xml:

Example 26.4. REST-Client configuration - pom.xml

```

<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j-rest</artifactId>
  <version>1.0.0.RELEASE</version>
</dependency>
```

Now, you set up the normal Spring Data Graph configuration, but point the database to an URL instead of a local directory, like so:

Example 26.5. REST client configuration - application context

```
<datagraph:config graphDatabaseService="graphDatabaseService"/>

<bean id="graphDatabaseService" class="org.neo4j.rest.graphdb.RestGraphDatabase">
    <constructor-arg value="http://localhost:7474/db/data/"/>
</bean>
```

Your project is now set up to work against a remote Neo4j Server.