Good Relationships

The Spring Data Graph Guidebook

Preface

1. Foreword: Rod Johnson, CEO of SpringSource

2. Foreword: Emil Eifrem, CEO of Neo Technology

3. 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 Library. Spring Data Graph is part of the Spring Data project which brings the convenient programming model of the Spring Framework to modern (mainly NoSQL) datastores. Spring Data Graph currently provides integration for the Neo4j Graph Database.

It was written by developers for developers. So hopefully we've created a documentation that is well received by our peers.

If you have any feedback to the Spring Data Graph Library or this book, please provide it via 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 narrative, that gives the reader an overview of the topics contained in the book. It contains lots of examples and more general discussion topics. This should be the only part of the book that is required to be read cover-to-cover.

We chose a tutorial describing the creation of a web applicaton (cineasts.net) that allows movie enthusiasts to find the favorites, rate them, connect with each other and enjoy social features. The application is running on Neo4j using Spring Data Graph and the well known Spring Web Stack.

The second part is the classic reference documentation containing the detailed information about the library. It discusses the programming model, the underlying assumptions, used toolset (like aspectj) as well as the APIs for the object-graph mapping and the template approach. The reference docs should be mainly used to look up concrete bits of information or to dig deeper into certain topics.

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. It uses a domain that should be familiar - movies. So for cineasts.net we decided to add a social touch to rating movies, allowing friends to share their scores and get recommendations for new friends and movies.

The tutorial walks 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. Of course the complete source code for the app is available at github.

Chapter 1. Allow me to introduce - Cineasts.net

Once upon a time we wanted to build a social movie database. First things first - we had a name: "Cineasts" - the cinema enthusiasts who are crazy about movies. So we went ahead and got the domain, cineasts.net and the project was almost complete.

We had some ideas about the domain too. Of course there should be actors who play roles in movies. We needed the Cineast, too, someone to rate the movies. And while they were there, they could also make friends. Find someone to accompany them to the cinema or share movie preferences. Even better, the engine behind all that should recommend new friends and movies to cineasts, based on their interests and existing friends.

When we looked for possible sources for data, IMDB was our first stop, but they're a little expensive for our tastes, charging 15k USD for data access. Fortunately we found TheMoviedb.org which provides user-generated data for free. The also have liberal terms and conditions and a nice API for fetching the data.

There were many more ideas but we wanted to get something done quickly. And this is how it should look.

Chapter 2. Scope: Spring

Being Spring developers, we would, of course, choose components of the Spring Framework to do most of the work. We'd already come up with the ideas - that should be enough.

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

But, wait, there is the new Spring Data project, started in 2010, which brings the convenience of the Spring programming model to NoSQL databases. That should fit our experience and help us to get started. We looked at the list of projects supporting the different NoSQL databases. Only one mentioned the kind of social network we were thinking of - Spring Data Graph for Neo4j, a graph database. Neo4j's pitch of "value in relationships" and the accompanying docs looked like what we needed. We decided to give it a try.

2.1. Preparations - Required Setup

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

Example 2.1. 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. 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 we were ready for the first spike: creating a simple MovieController showing a static view. Check. Next was the setup for Spring Data Graph. We looked at the README at github and then checked it with the manual. Quite a lot of Maven setup for AspectJ but otherwise not so much to add. Time to add a few lines to our Spring configuration.

Example 2.3. applicationContext.xml

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

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

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

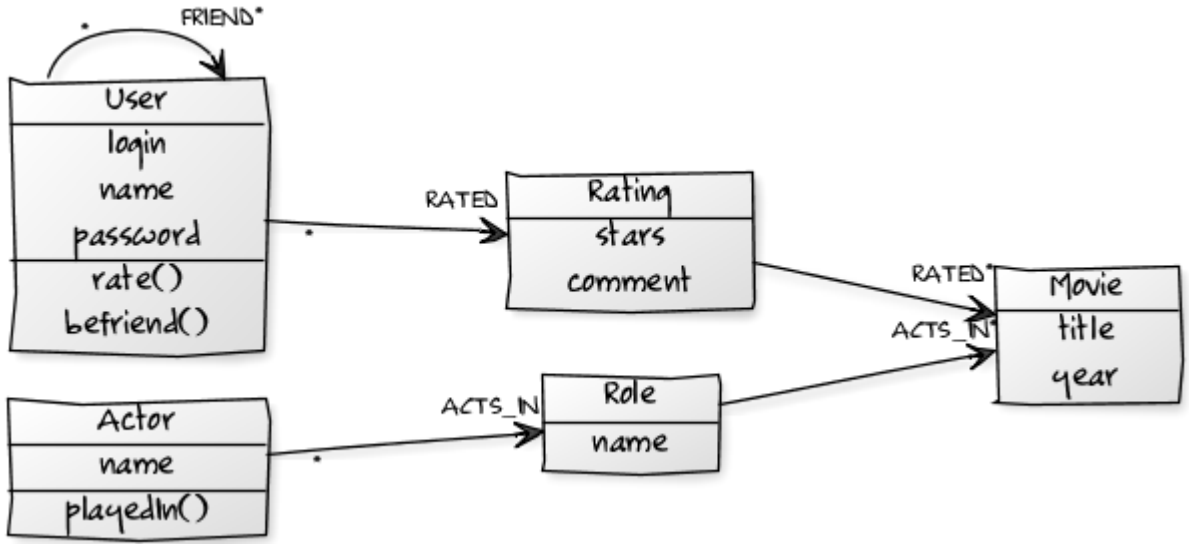
Example 2.4. dispatcherServlet-servlet.xml

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

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

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

We spun up Jetty to see if there were any obvious issues with the config. It all seemed to work just fine. Check.

Chapter 3. Setting the Stage - Movies Domain

The domain model was the next thing we planned to work on. We wanted to sketch it out first before diving into library details. We also looked at the datamodel of core themoviedb data to confirm that it matched our expectations.



In Java code this looks pretty straightforward:

```
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;
```

```
}
```

We then wrote some tests to show the basic plumbing works.

Chapter 4. Graphs ahead - Learning Neo4j

Now came the unknown - how to put these domain objects into the graph. First we read up about graph databases, especially Neo4j. The Neo4j datamodel consists of nodes and relationships, both of which can have properties. Relationships are first class citizens in Neo4j, meaning we can link together nodes into semantically rich networks - we really liked that. Then we found we could index nodes and relationships by {name, value} pairs to quickly get hold of them as starting points for further processing. We also found we could imperatively traverse of relationships using the core API, and in a declarative way using a query-like Traversal Description.

We also learned that Neo4j was fully transactional and completely upholds ACID guarantees for out data. This is unusual for NoSQL databases, but easier for us to get our head around than non-transactional eventual consistency. It also makes us feel safe, though it also means that we had to manage transactions. Keep that in mind.

Initially we used the core Neo4j API to get a feeling for that. And also to see, how (probably) the domain might look when it's saved in the graph store. After adding the Maven dependency, it was ready to go.

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

```
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. Conjuring Magic - Spring Data Graph

That was the pure graph database. Using this in our domain would pollute our classes with lots of graph database details. We don't want that. Spring Data Graph promised to do the heavy lifting for us. So we checked that next. Spring Data Graph depends heavily on AspectJ magic. Some parts of our classes would behave differently, but it would not be visible in our code. We were going to give it a try.

First step was lots of 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.M5</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 configuration was much easier, thanks to a provided namespace.

```
<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. Decorations - Annotated Domain

Looking at the documentation again, we found a simple Hello-World example and tried to understand it. The entities were annotated with @NodeEntity, that was simple, so we added the annotation to our domain classes too. Relationships got their own annotation named @RelationshipEntity. Property fields are taken care of automatically.

It's time to put this to a test. How can we be assured that a field is persisted to the graph store? There seemed to be two possibilities. First was to get a GraphDatabaseContext injected and use its getById() method. The other one was a Finder approach. But let's try to keep things simple. How can we persist an entity and how to get its id? Looking at the documentation revealed that there are a bunch of methods introduced to the entities by the aspects. That's not obvious, but we found the two that would help here - entity.persist() and entity.getNodeId().

So our test looked like this.

```
@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());
}
```

That worked! But what about transactions? We didn't declare the test to be transactional. After further reading we learned that persist() creates an implicit transaction - so that was like an EntityManager would behave. Ok, now we're getting somewhere. We also learned that for more complex operations on the entities we'd need external transactions.

Chapter 7. Do I know you? - Indexing

There an @Indexed annotation for fields. We wanted to try this out, and use it to guide the next test. We added an @Indexed to the id field of the movie. This field is intended to represent the external id that will be used in URIs and will stable over database imports and updates. This time we went with the Finder to retrieve the indexed movie.

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

@Autowired FinderFactory finderFactory;

@Test [@Transactional] public void persistedMovieShouldBeRetrievableFromGraphDb() {
    int id=1;
    Movie forrestGump = new Movie(id, "Forrest Gump", 1994).persist();
    NodeFinder<Movie> movieFinder = finderFactory.createNodeEntityFinder(Movie.class);
    // REMINDER, the "null" stands for an optional index name
    Movie retrievedMovie = movieFinder.findByPropertyValue(null, "id",id);
    assertEqual("retrieved movie matches persisted one",forrestGump,retrievedMovie);
    assertEqual("retrieved movie title matches","Forrest Gump",retrievedMovie.getTitle());
}
```

Surprisingly, this failed with an exception about not being in a transaction, which means we forgot to add the @Transactional annotation. That's easy enough to add to the test, and resume the test/code cycle.

Chapter 8. Serving a good cause - Repository

That was the first method to add to the brand new repository. So we created a repository for the application, annotated it with @Repository and @Transactional. We did the same for the Actor.

```
@Repository @Transactional
public class CineastsRepostory {
  FinderFactory finderFactory;
  Finder<Movie> movieFinder;
  @Autowired
  public CineastsRepostory(FinderFactory finderFactory) {
      this.finderFactory = finderFactory;
      this.movieFinder = finderFactory.createNodeEntityFinder(Movie.class);
  }
  public Movie getMovie(int id) {
      return movieFinder.findByPropertyValue(null,"id", id);
  }
}
```

Chapter 9. A convincing act - Relationships

9.1. Value in Relationships - Creating them

Next were relationships. Direct relationships didn't require any annotation. Unfortunately we had none of those, because ours had more semantics. So we went for 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:

```
@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 new, but got an exception saying that this is not allowed. This must be a strange restriction about having only correctly constructed RelationshipEntities. To fix it, we had to recall the relateTo method from the introduced methods on the NodeEntities. After checking it turned out to be exactly what we needed. We then added the method for connecting movies and actors to the actor - which seems a more natural fit.

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

9.2. Who's there ? - Accessing related entities

What was left? Accessing those relationships. We already had the appropriate fields in both classes. Time to annotate them correctly. For the fields providing access to the entities on the each side of the relationship this was straightforward. Providing the target type again (thanks to Java's type erasure) and the relationship type (learned from the Neo4j lesson before) there was only the direction left. Which defaults to OUTGOING so only for the movie we had to specify it.

```
@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> cast;

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

While reading about those relationship-sets we learned that they are handled by managed collections of Spring Data Graph. So whenever we add something to the set or remove it, it automatically reflects that in the underlying relationships. Neat. But this also meant we mustn't initialize the fields. Something we will certainly forget not to do in the future, so watch out for it.

We made sure to add a test for those, so are assured that the collections worked as advertised (and also ran into the intialization problem above).

9.3. May I introduce ? - Accessing Relationships themselves

But we still couldn't access the Role relationships. There was more to read about this. For accessing the relationship in between the nodes there was a separate annotation @RelatedToVia. And we had to declare the field as readonly Iterable<Role>. That should make sure that we never tried to add Roles (which I couldn't create on my own anyway) to this field. Otherwise the annotation attributes were similar to those used for @RelatedTo. So off we went, creating our first real relationship (just kidding).

```
@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 the tests proved that those relationship fields really mirrored the underlying relationships in the graph and instantly reflected additions and removals we were pretty satisfied with our domain.

Chapter 10. Curtains Up! - Get it running

10.1. Requisites - Populating the database

Time to put this on display. But we needed some test data first. 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.

```
@Service
public class DatabasePopulator {

  @Autowired GraphDatabaseContext ctx;
  @Autowired MoviesRepository repository;

  @Transactional
  public List<Movie> populateDatabase() {
      Actor tomHanks = new Actor("1", "Tom Hanks").persist();
      Movie forestGump = new Movie("1", "Forrest Gump").persist();
      forestGump.setYear(1994);
      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";
  }
}
```

```
<%@ 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>
```

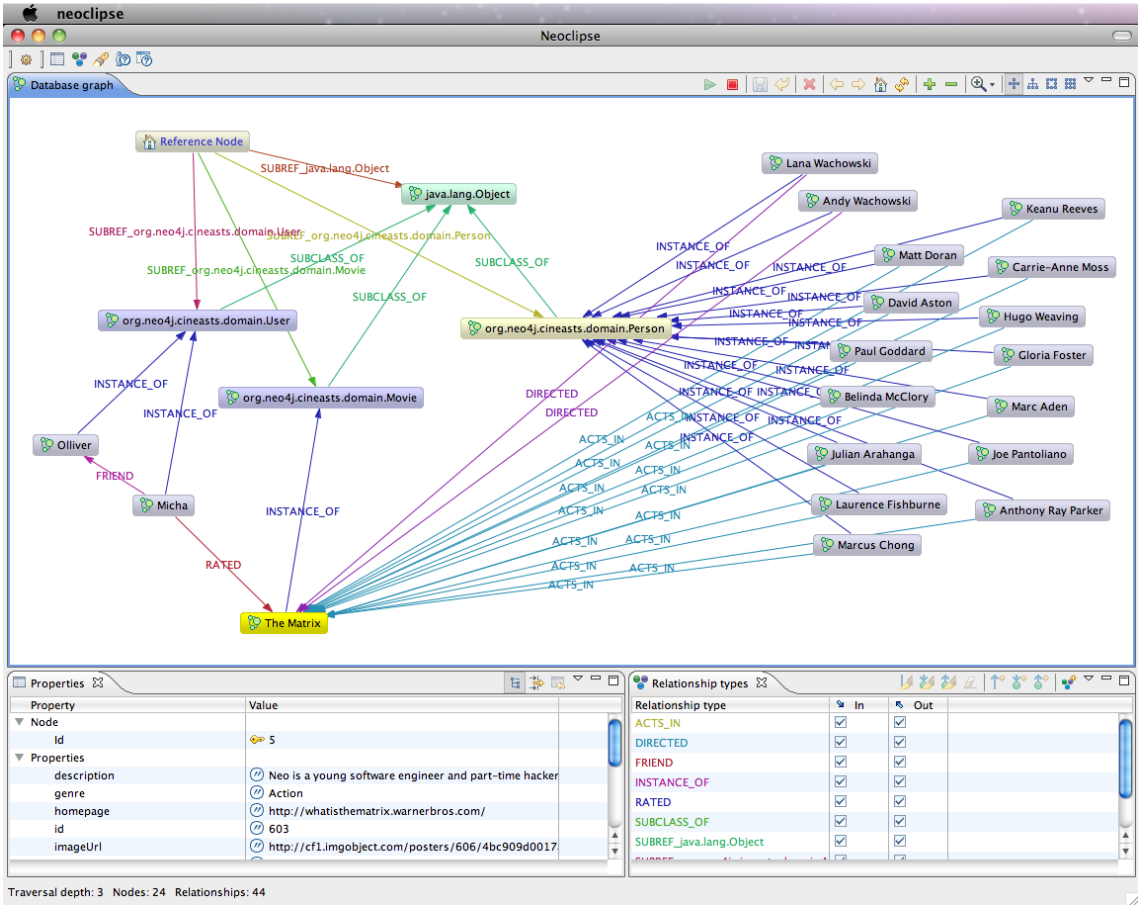
```
  </c:when>
  <c:otherwise>
      No Movie with id ${id} found!
  </c:otherwise>
</c:choose>
```

See the misused GET parameter for that (don't do this at home, the REST guys will be upset). This is only for running it from the browser address line. Better use POST and curl for the call. So we called the URI and it showed the single added movie on screen.

10.2. Behind the scenes - Peeking at the Datastore

10.2.1. Eye candy - Neoclipse visualization

After filling the database we wanted to see what the graph looked like. So we checked out two tools that are available for inspecting the graph. First Neoclipse, an eclipse RCP application or plugin that connects to existing graph stores and visualizes their content. After getting an exception about concurrent access, I learned that I have to use Neoclipse in readonly mode when my webapp had an active connection to the store. Good to know.



10.2.2. Hardcore "Hacking" - Neo4j Shell

Besides our movies and actors connected by ACTS_IN relationships there were some other nodes. The reference node which is an automatically provided "root node" in Neo4j and can be used to anchor subgraphs for easier access. And Spring Data Graph also represented the type hierarchy of my entities in the graph. Obviously for some internal housekeeping and type checking.

For console junkies there is also a shell that can reach into a running neo4j store (if that one was started with enableRemoteShell) or provide readonly access to a graph store directory.

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

It uses some shell metaphors like cd and ls to navigate the graph. There are also more advanced commands like using indexes and traversals. I tried to play around with them in this shell sesson.

```
neo4j-sh[readonly] (0)$ ls
(me) --[SUBREF_java.lang.Object]-> (3)
(me) --[SUBREF_org.neo4j.cineasts.domain.Movie]-> (6)
(me) --[SUBREF_org.neo4j.cineasts.domain.Person]-> (8)
(me) --[SUBREF_org.neo4j.cineasts.domain.User]-> (2)
neo4j-sh[readonly] (0)$ cd 6
neo4j-sh[readonly] (6)$ ls
*class =[org.neo4j.cineasts.domain.Movie]
*count =[39]
(me) <-[INSTANCE_OF]-- (The Matrix Revolutions,123)
(me) <-[INSTANCE_OF]-- (The Matrix Reloaded,110)
(me) <-[INSTANCE_OF]-- (The Matrix,93)
...
neo4j-sh[readonly] (6)$ cd 93
neo4j-sh[readonly] (The Matrix,93)$ ls
*description  =[Neo is a young software engineer and part-time hacker who is singled out by some myste
*genre        =[Action]
*homepage     =[http://whatisthematrix.warnerbros.com/]
*id           =[603]
*imageUrl     =[http://cf1.imgobject.com/posters/606/4bc909d0017a3c57fe003606/the-matrix-mid.jpg]
*imdbId       =[tt0133093]
*language     =[en]
*lastModified =[1299968642000]
*releaseDate  =[922831200000]
*runtime      =[136]
*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,109)
...
(me) <-[ACTS_IN]-- (Keanu Reeves,96)
(me) <-[DIRECTED]-- (Andy Wachowski,95)
(me) <-[DIRECTED]-- (Lana Wachowski,94)
(me) --[INSTANCE_OF]-> (6)
(me) <-[RATED]-- (Micha,1)
```

Chapter 11. Showing off - Web views

After we had the means to put some data in the graph database, we also wanted to show it. So adding the controller method to show a single movie with its attributes and cast in a jsp was straightforward. Actually just using the repository to look the movie up and add it to the model. Then forward to the /movies/show view and voilá.

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

Later the nice UI would look like that:

11.1. What was his name? - Searching

The next thing was to allow users to search for some movies. So we needed some fulltext-search capabilities. As the index provider implementation of Neo4j builds on lucene we were delighted to see that fulltext indexes are supported out of the box.

We happily annotated the title field of my Movie class with @Index(fulltext=true) and was told with an exception that we have to specify a separate index name for that. So it became @Indexed(fulltext = true, indexName = "search"). The corresponding finder method is called findAllByQuery. So there

was our second repository method for searching movies. To restrict the size of the returned set we just added a limit for now that truncates the result after so many entries.

```
public void List<Movie> searchForMovie(String query, int count) {
    List<Movie> movies=new ArrayList<Movie>(count);
    for (Movie movie : movieFinder.findAllByQuery("title", query)) {
        movies.add(movie);
        if (count-- == 0) break;
    }
    return movies;
}
```

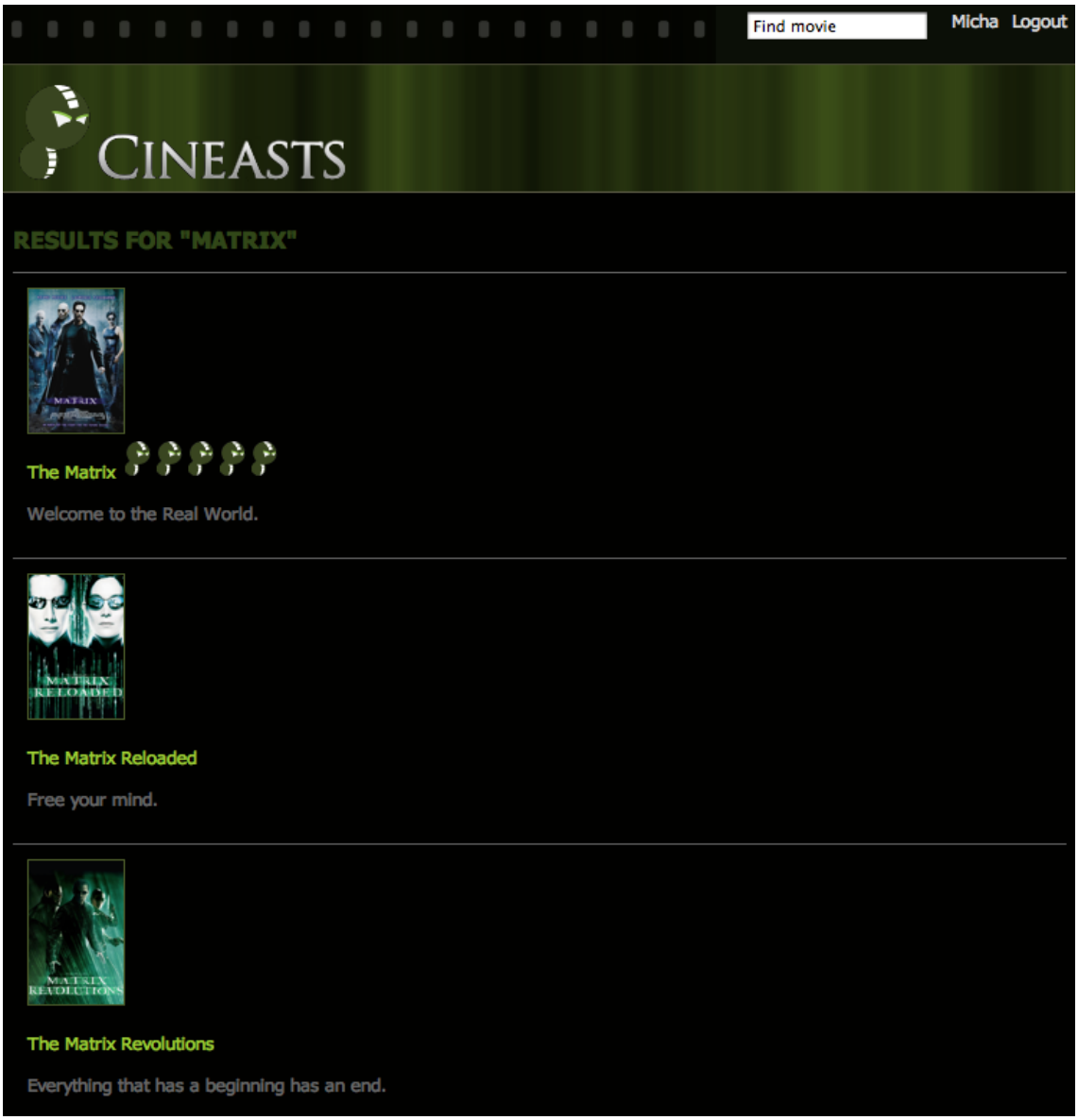
11.2. Look what we've found - Listing Results

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 was accessed by the getters in the domain classes.

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

```
<h2>Movies</h2>

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

Here is another teaser, what the final UX would look like for that:


Find movie
Micha Logout
CINEASTS
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. Movies 2.0 - Adding social

But this was just a plain old movie database (POMD). Our idea of socializing this business wasn't yet realized.

12.1. Look, mom a Cineast! - Users

So we took the User class that we'd already coded and made it a full fledged Spring Data Graph member. We added the ability to make friends and to rate movies. With that there was also a simple UserRepository that was able to look up users by id.

```
@NodeEntity
class User {
    @Indexed
    String login;
    String name;
    String password;
    @RelatedTo(elementClass=Movie.class, type="RATED")
    Set<Rating> ratings;

    @RelatedTo(elementClass=User.class, type="FRIEND")
    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 my DatabasePopulator to add some users and ratings to the initial setup.

```
@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).persi
    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. Beware, Critics - Rating

We also put a ratings field into the movie to be able to show its ratings. And a method to average its star rating.

```
class Movie {
    @RelatedToVia(elementClass=Rating.class, type="RATED", direction = Direction.INCOMING)
    Iterable<Rating> ratings;

    public int getStars() {
        int stars, int count;
        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. Next steps were to add this information to the UI of movie and create a user profile page. But for that to happen they must be able to log in.

Chapter 13. Protecting Assets - Adding Security

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 only worked with a valid user account had to be secured as well.

We used Spring Security for that, writing 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. pom.xml for spring-security

```
        <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. web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/applicatioContext-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. 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

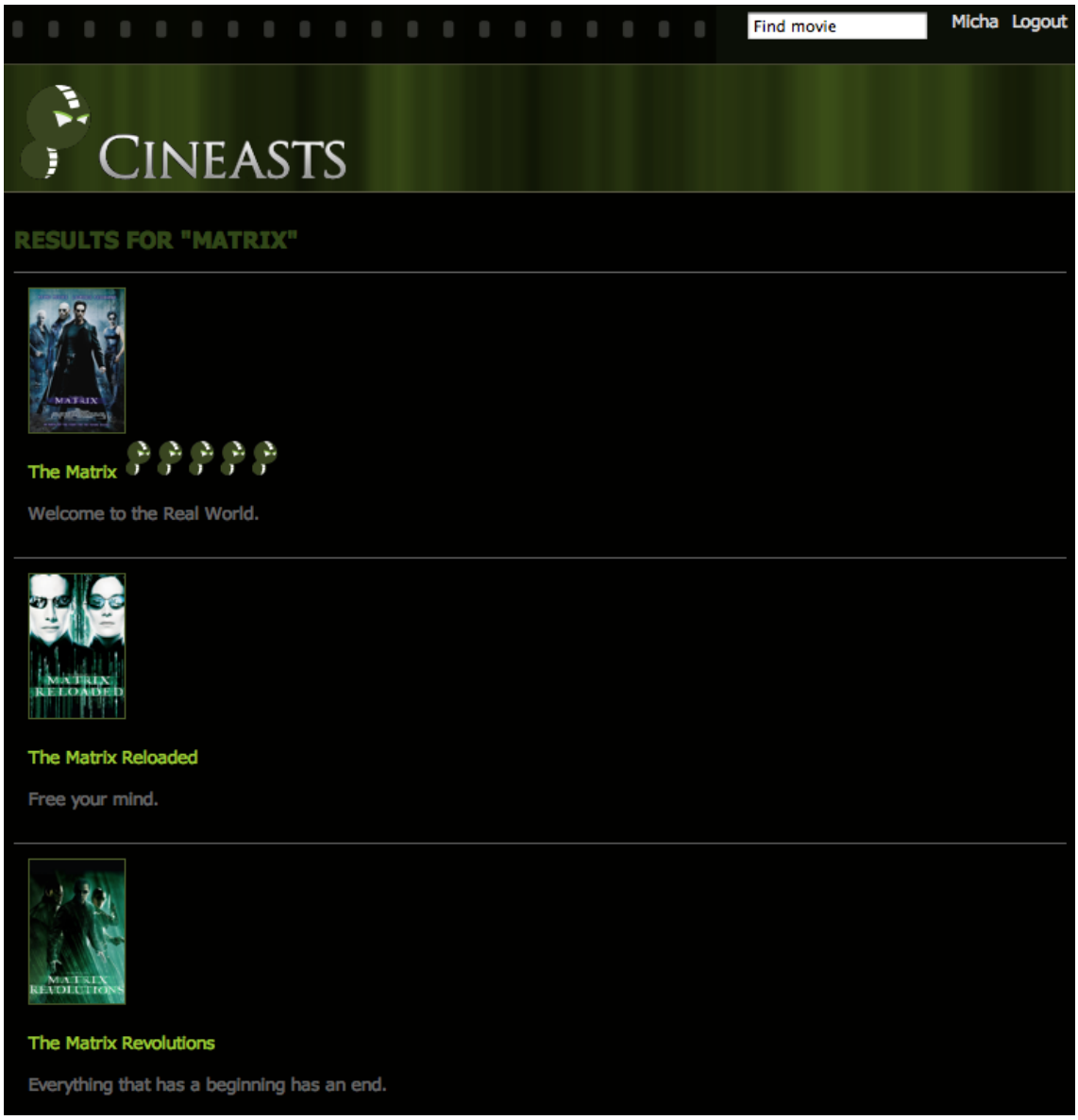
```
@Service
public class CineastsUserDetailsService implements UserDetailsService, InitializingBean {

    @Autowired private FinderFactory finderFactory;
    private NodeFinder<User> userFinder;

    @Override
    public void afterPropertiesSet() throws Exception {
        userFinder = finderFactory.createNodeEntityFinder(User.class);
    }

    @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 userFinder.findByPropertyValue("users","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;
    }
}
```

After that a logged in user was available in the session and could so be used for all the social interactions. Most of the work done next was 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. As a teaser we'd like to show off the user profile page, as it will be rendered after UX heavy lifting.

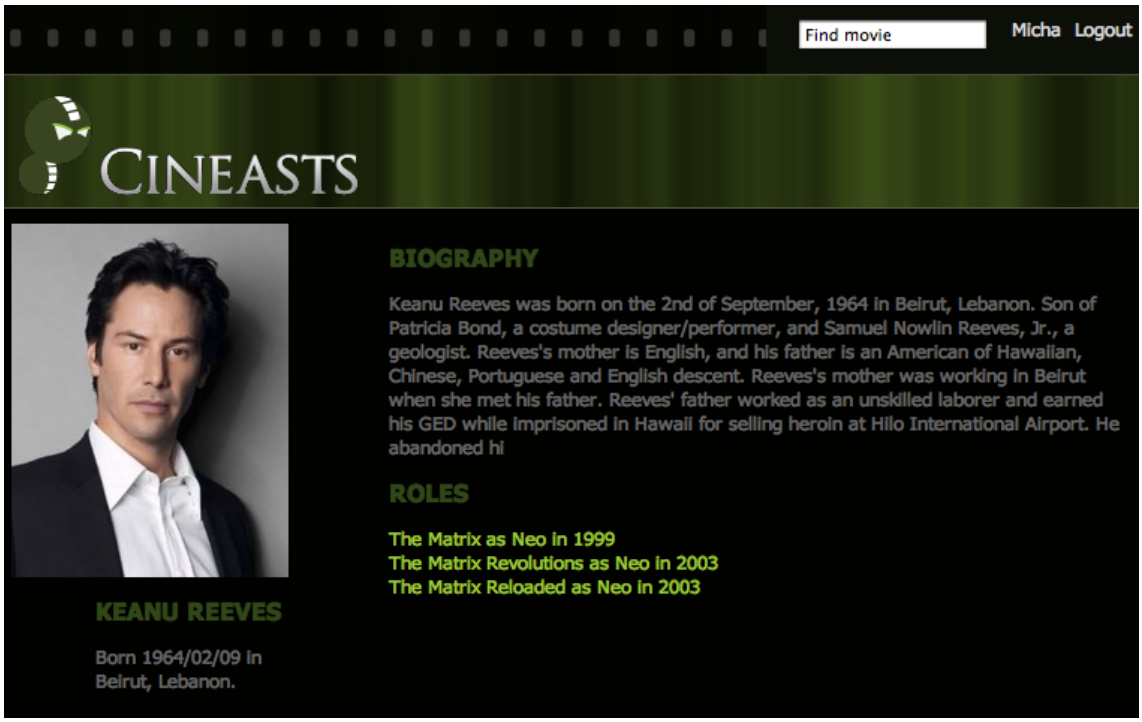
Chapter 14. Oh the Glamour - More UI

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 UX people involved and the results were impressive. This sections presents some of the remaining screenshots of cineasts.net.

Some of the noteworthy things. As Spring Data Graph does a read-through to the datastore 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.


Find movie
Micha Logout
CINEASTS
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
RELEASE INFO
Released 1999/31/03
Homepage
Studio Warner Bros. Pictures
TRAILERS
YouTube
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. The dusty archives - Importing Data

Then it was time to pull the data from themoviedb.org. Registering there and getting an API key was simple, using the API on the command line with curl too. 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.

```
   [{"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"}]
```

```
    [{"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 insert 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.

```
@Transactional
public Movie importMovie(String movieId) {
    Movie movie = moviesRepository.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 four
    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 were a limiting factor. As soon as the data was stored locally it took only subseconds to create the data in the Neo4j graph database.

Chapter 16. Movies! Friends! Bargains! - Recommendations

In the last part of this exercise we wanted to add recommendations to the app. One obvious recommendation is movies that our friends liked (and their friends too, but with less importance). The second was recommendations for new friends that also liked the movies that we liked most.

Doing this kind of ranking algorithms is really fun with graph databases. They are applied to the graph by traversing it in a certain order, collecting information on the go and deciding which paths to follow and what to include in the results.

Lets say I'm only interested in the top 10 recommendations each.

```
        // TODO Work In Progress 1/path.length()*stars
        user.breathFirst().relationship(FRIEND, OUTGOING).relationship(RATED, OUTGOING).evaluate(new E
            if (path.length > 5) return EXCLUDE_AND_STOP;
            Relationship rating = path.lastRelationship();
            if (rating.getType().equals(RATED)) {
                rating.getProperty()
                return INCLUDE_AND_STOP;
            }
            return INCLUDE_AND_CONTINUE;
        })
```

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 applies core Spring concepts to the development of solutions using a graph style data store. The basic approach is to mark simple POJO entities with Spring Data Graph annotations. That enables the AspectJ aspects that are contained with the framework to adapt the instantiation and field access to have them stored and retrieved from the graph store. Entities are mapped to nodes of the graph, references to other entities are represented by relationships. There are also special relationship entities that provide access to the properties of graph relationships.

For the developer of a Spring Data Graph backed application only the public annotations are relevant, basic knowledge of graph stores is needed to access advanced functionality like traversals. Traversal results can also be mapped to fields of entities.

Chapter 17. Introduction to Neo4j

Neo4j is a graph database. It is a fully ACID transactional database that stores data structured as graphs. A graph consists of nodes, connected by relationships. It is a flexible data structure that allows for high query performance on complex data, while being intuitive for the developer.

Neo4j has been in commercial development for 10 years and in production for over 7 years. It is a mature and robust graph database that:

- has an intuitive graph-oriented model for data representation. Instead of tables, rows, and columns, you work with a flexible graph network consisting of nodes, relationships, and properties.
- has a disk-based, native storage manager completely optimized for storing graph structures for maximum performance and scalability.
- is scalable. Neo4j can handle graphs of several billion 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 fast traversals in the node space.
- can be deployed as a standalone server or an embedded database with a very small footprint (~700k jar).
- has a simple and convenient API.

In addition, Neo4j includes the usual database features: ACID transactions, durable persistence, concurrency control, transaction recovery, high availability and everything else you'd expect from an enterprise database. Neo4j is released under a dual free software/commercial license model.

17.1. 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 relationship and allows high performance traversal of those structures. With property graphs it is possible to add an arbitrary number of properties to nodes and relationships.

17.2. 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 an IndexManager, database lifecycle callbacks, transation management and more.

The EmbeddedGraphDatabaseService is an implementation of GraphDatabaseService that is used to embed Neo4j in a Java application. This implmentation is used so as to provide the highest and tightest integration. There are other, remote implementations that provide access to Neo4j stores via REST.

17.3. Creating Nodes and Relationships

Using the API of GraphDatabaseService it is easy to create nodes and relate them to each other. Relationships are named. Both nodes and relationships can have properties. Property values can be primitive Java types and Strings, byte arrays for binary data, or arrays of other 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.

```
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();
}
```

17.4. 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 means via a concise DSL to define TraversalDescriptions that can then be applied to a start node and will produce a stream of nodes and/or relationships as a lazy result using an Iterable.

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

17.5. Indexing

The best way for retrieving start nodes for traversals is using Neo4j's index facilities. The GraphDatabaseService provides access to the IndexManager which in turn retrieves named indexes for nodes and relationships. Both can be indexed with property names and values. Retrieval is done by query methods on Index to return an IndexHits iterator.

```
IndexManager indexManager = graphDb.index();
Index<Node> nodeIndex = indexManager.forNodes("a-node-index");
nodeIndex.add(node, "property","value");
for (Node foundNode = nodeIndex.get("property","value")) {
    assert node.getProperty("property").equals("value");
}
```

Note: Spring Data Graph provides auto-indexing via the @Indexed annotation, while this still is a manual process when using the Neo4j API.

Chapter 18. Programming model for Spring Data Graph

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.

18.1. Overview of the AspectJ support

Behind the scenes Spring Data Graph leverages AspectJ aspects to modify the behavior of simple POJO entities to be able to be backed by a graph store. Each entity is backed by a node that holds its properties and relationships to other entities. AspectJ is used to intercept field access and to reroute it to the backing state (either its properties or relationships). For relationship entities the fields are similarly mapped to properties. There are two specially annotated fields for the start and the end node of the relationship.

The aspect introduces some internal fields and some public methods to the entities for accessing the backing state via `getPersistentState()` and creating relationships with `relateTo` and retrieving relationship entities via`getRelationshipTo`. It also introduces finder methods like `find(Class<? extends NodeEntity>, TraversalDescription)` and equals and hashCode delegation.

Spring Data Graph internally uses an abstraction called EntityState that the field access and instantiation advices of the aspect delegate to, keeping the aspect code very small and focused to the pointcuts and delegation code. The EntityState then uses a number of FieldAccessor factories to create a FieldAccessor instance per field that does the specific handling needed for the concrete field.

18.2. Using annotations to define POJO entities and relationships

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

18.2.1. @NodeEntity: The basic building block

The `@NodeEntity` annotation is used to declare a POJO entity to be backed by a node in the graph store. Simple fields on the entity are mapped by default to properties of the node. Object references to other NodeEntities (whether single or Collection) are mapped via relationships. If the annotation parameter `useShortNames` is set to false, the properties and relationship names used will be prepended with the class name of the entity. If the parameter `fullIndex` is set to true, all fields of the entity will be indexed. If the `partial` parameter is set to true, this entity takes part in a cross-store setting where only the parts of the entity not handled by JPA will be mapped to the graph store.

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

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

18.2.2. @RelatedTo: Connecting NodeEntities

Relationships to other NodeEntities are mapped to graph relationships. Those can either be single relationships (1:1) or multiple relationships (1:N). In most cases single relationships to other node entities don't have to be annotated as Spring Data Graph can extract all necessary information from the field using reflection. In the case of multiple relationships, the `elementClass` parameter of @RelatedTo must be specified because of type erasure. The `direction` (default OUTGOING) and `type` (inferred from field name) parameters of the annotation are optional.

Relationships to single node entities are created when setting the field and deleted when setting it to null. For multi-relationships the field provides a managed collection (Set) that handles addition and removal of node entities and reflects those in the graph relationships.

```
@NodeEntity
public class Movie {
 private Actor topActor;
}
@NodeEntity
public class Person {
 @RelatedTo(type = "topActor", direction = Direction.INCOMING)
 private Movie wasTopActorIn;
}
@NodeEntity
public class Actor {
 @RelatedTo(type = "ACTS_IN", elementClass = Movie.class)
 private Set<Movie> movies;
}
```

18.2.3. @RelationshipEntity: Rich relationships

To access the full data model of graph relationships, POJOs can also be annotated with @RelationshipEntity. Relationship entities can't be instantiated directly but are rather accessed via node entities, either by @RelatedToVia fields or by the `relateTo` or `getRelationshipTo` methods. Relationship entities may contain fields that are mapped to properties and two special fields that are annotated with @StartNode and @EndNode which point to the start and end node entities respectively. These fields are treated as read only fields.

```
@RelationshipEntity
public class Role {
 @StartNode
 private Actor actor;
 @EndNode
 private Movie movie;
}
```

18.2.4. @RelatedToVia: Connecting NodeEntitites via RelationshipEntities

To provide easy programmatic access to the richer relationship entities of the data model a different annotation @RelatedToVia can be declared on fields of Iterables of the relationship entity type. These Iterables then provide read only access to instances of the entity that backs the relationship of this relationship type. Those instances are initialized with the properties of the relationship and the start and end node.

```
@NodeEntity
```

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

18.2.5. @StartNode: Starting NodeEntity of RelationshipEntity

Annotation for the start node of a relationship entity, read only.

18.2.6. @EndNode: Ending NodeEntity of RelationshipEntity

Annotation for the end node of a relationship entity, read only.

18.2.7. @Indexed: Making entities searchable by field value

The @Indexed annotation can be declared on fields that are intended to be indexed by the Neo4j IndexManager, triggered by value modification. The resulting index can be used to later retrieve nodes or relationships that contain a certain property value (for example a name). Often an index is used to establish the start node for a traversal. Indexes are accessed by a Finder for a particular NodeEntity or RelationshipEntity, created via a FinderFactory.

GraphDatabaseContext exposes the indexes for Nodes and Relationships. Indexes can be named, for instance to keep separate domain concepts in separate indexes. That's why it is possible to specifiy an index name with the @Indexed annotation. It can also be specified at the entity level, this name is then the default index name for all fields of the entity. If no index name is specified, it defaults to the one configured with Neo4j ("node" and "relationship").

18.2.8. @GraphTraversal

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 NodeEntities that are the result of a traversal starting at the current NodeEntity. The TraversalDescription used for this is created by a TraversalDescriptionBuilder whose class is referred to by the `traversalBuilder` attribute of the annotation. The class of the expected NodeEntities is provided with the `elementClass` attribute.

18.2.9. @GraphProperty: Cross-store persisted fields

It is not necessary to annotate 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. Transient fields are not persisted. This annotation is mainly used for cross-store persistence.

18.3. Indexing

The Neo4j graph database can use different index providers for exact lookups and fulltext searches. Lucene is used as a index provider implementation. There is support for distinct indexes for nodes and relationships which can be configured to be of fulltext or exact types.

Using the standard Neo4j API, Nodes and Relationships and their indexed field-value combinations have to be added manually to the appropriate index. When using Spring Data Graph, this task is simplified by eased by applying an `@Indexed` annotation on entity fields. This will result in updates to the index on every change. 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 can also

set the index-name to be used. If @Indexed annotates the entity class, the index-name for the whole entity is preset to that value. Not providing index names defaults them to "node" and "relationship" respectively.

Query access to the index happens with the Node- and RelationshipFinders that are created via an instance of `org.springframework.data.graph.neo4j.finder.FinderFactory`. 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).

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

    // automatically indexed numerically
    @Indexed
    int age;

}

@NodeEntity
@Indexed(indexName="groups")
class Group {
    @Indexed
    String name;

    @RelatedTo(elementClass = Person.class, type = "people" )
    Set<Person> people;
}

NodeFinder<Person> finder = finderFactory.createNodeEntityFinder(Person.class);

// exact finder
Person mark = finder.findByProperyValue("people","name","mark");

// numeric range queries
for (Person middleAgedDeveloper : finder.findAllByRange(null, "age", 20, 40)) {
    Developer developer=middleAgedDeveloper.projectTo(Developer.class);
}
```

Neo4jTemplate also offers index support, providing auto-indexing for fields at creation time of nodes and relationships. 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 / query expression and push the results wrapped uniformly as Paths to the supplied `PathMapper` to be converted or collected.

18.4. Finding nodes with finders

Spring Data Graph also comes with a type bound Repository-like Finder implementation that provides methods for locating nodes and relationships:

- using direct access `findById(id)`,
- iterating over all nodes of a node entity type (findAll),
- counting the instances of a node entity type (count),

- iterating over all indexed instances with a certain property value (findAllByPropertyValue),
- getting a single instance with a certain property value (findByPropertyValue),
- iterating over all indexed instances within a certain numerical range (inclusive) (findAllByRange),
- iterating over a traversal result (findAllByTraversal).

The Finder instances are created via a FinderFactory to be bound to a concrete node or relationship entity class. The FinderFactory is created in the Spring context and can be injected.

```
NodeFinder<Person> finder = finderFactory.createNodeEntityFinder(Person.class);
Person dave=finder.findById(123);
int people = finder.count();
Person mark = finder.findByPropertyValue("name", "mark");
Iterable<Person> devs = finder.findAllByProperyValue("occupation","developer");
Iterable<Person> davesFriends = finder.findAllByTraversal(dave,
    Traversal.description().pruneAfterDepth(1)
    .relationships(KNOWS).filter(returnAllButStartNode()));
```

18.5. Transactions in Spring Data Graph

Neo4j is a transactional datastore which only allows modifications within transaction boundaries and fullfills the ACID properties. Reading from the store is also possible outside of 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. Note: The explicit XML configuration given below is encoded in the `Neo4jConfiguration` configuration bean that uses Spring's @Configuration functioanlity. This simplifies the configuration. An example is shown further below.

```
<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 running multiple transactional resources there are two options. First of all you can have Neo4j participate in the externally set up transaction manager using the new SpringProvider by enabling the configuration parameter for your graph database. Either via the spring config or the configuration file (neo4j.properties).

```
<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"/>
```

You can configure a stock XA transaction manager to be used with Neo4j and the other resources (e.g. Atomikos, JOTM, App-Server-TM). For a bit less secure but fast 1 phase commit best effort, use the implementation coming with Spring Data Graph (`ChainedTransactionManager`). 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.

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

18.6. Session handling - attached and detached entities

By default newly created node entities are in a detached state. When `persist()` is called on the entity it is attached to the graph store and its properties and relationships are persisted as well. Changing an attached entity inside a transaction will write through the changes to the datastore. Whenever an entity is changed outside of a transaction it will be considered detached. The changed data is stored in the entity itself and not written back to the datastore.

All entities that are returned by library functions are initially in an attached state. Changing them outside of a transaction detaches them. For writing the changes back it is necessary to `persist()` them again.

Persisting an entity not only persists that single entity but will traverse its existing and new relationships and persist the cluster of detached entities that it is part of. The borders of this cluster are formed by

attached entities. The persist operation creates its own, implicit transaction. When it is called withina external transaction it participates otherwise it is an atomic operation.

Please keep in mind that the session handling behaviour is still heavily developed. The defaults and also other aspects of the behaviour are likely to change in subsequent releases. At the moment there is no support for the creation of relationships outside of transactions and also more complex operations like creating whole subgraphs outside of transactions is not supported.

```
@NodeEntity
class Person {
    String name;
}
Person p = new Person().persist();
```

18.7. Reified types for entities

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 `NodeTypeStrategy` 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, count, etc.

There are three available implementations to choose from.

- `IndexingNodeTypeStrategy`

  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.

- `SubReferenceNodeTypeStrategy`

  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.

- `NoopNodeTypeStrategy`

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

The default implementation is `IndexingNodeTypeStrategy` for new graphs. If using an existing graph, Spring Data Graph will default to the strategy first used when the graph was created.

18.8. Methods added to entity classes

The node and relationship aspects introduce (via ITD - inter type declaration) several methods to the entities that make common tasks easier. Unfortunately these methods are not generified yet, so the results have to be casted to the correct return type.

persisting the node-entity initially and after changes outside of a transaction, persist participates in a transaction or creates its own implict transaction.

```
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(targetEnttiy, 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 relationship and index entries

```
entity.remove()
```

projecting to a different target type

```
entity.projectTo(targetClass)
```

traversing, starting at the current node

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

18.9. Dynamic typing - Projection to unrelated, fitting types

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.

```
// not related to Person at all
@NodeEntity
class Trainee {
    String name;
    @RelatedTo(elementClass=Training.class);
    Set<Training> trainings;
}

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

18.10. Neo4jTemplate

The `Neo4jTemplate` offers the convenient API of Spring templates for the Neo4j graph database. There are methods for creating nodes and relationships that automatically set provided properties and optionally index certain fields. Other methods (`index`, `autoindex`) will index them.

For the querying operations Neo4jTemplate unifies the result with the `Path` abstraction that comes from Neo4j. Much like a resultset a path contains `nodes()` and `relationships()` starting at a `startNode()` and ending with `aendNode()`, 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 Java objects.

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.

Traversal methods are the bread and butter of graph operations. As such, they are fully supported in the `Neo4jTemplate`. The `traverseNext` method traverses to the direct neighbours of the start node filtering the relationships according to its parameters.

The `traverse` method covers the full fledged traversal operation that takes a powerful `TraversalDescription` (most probably built from the `Traversal.description()` DSL) and runs it from the start node. Each path that is returned via the traversal is passed to the `PathMapper` to be processed accordingly.

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.

```
Neo4jOperations neo = new Neo4jTemplate(grapDatabase);
Node michael = neo.createNode(_("name","Michael"),"name");
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");
```

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

18.11. Bean Validation - JSR-303

Spring Data Graph supports property based validation support. So whenever a property is changed, it is checked against the annotated constraints (.e.g @Min, @Max, @Size, etc). Validation errors throw a ValidationException. For evaluating the constraints the validation support that comes with Spring is used. To use it a validator has to be registered with the GraphDatabaseContext, if there is none, no validation will be performed (any registered Validator or (Local)ValidatorFactoryBean will be used).

```

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

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

Chapter 19. Setup required for Spring Data Graph

To use Spring Data Graph in your application, some setup is required. For building the application the necessary Maven dependencies must be included and for the AspectJ weaving some extensions of the compile goal are necessary. This chapter also discusses the Spring configuration needed to set up Spring Data Graph. Examples for this setup can be found in the Spring Data Graph examples.

19.1. Maven Configuration

As stated in the requirements chapter, Spring Data Graph projects are easiest to build with Apache Maven. The main dependencies are Spring Data Graph itself, Spring Data Commons, some parts of the Spring Framework and of course the Neo4j graph database.

19.1.1. Repositories

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

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

19.1.2. Dependencies

The dependency on `spring-data-neo4j` should transitively pull in 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.

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j</artifactId>
  <version>1.0.0.M5</version>
</dependency>

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

19.1.3. AspectJ build configuration

As Spring Data Graph uses AspectJ for build time aspect weaving of your entities, it is necessary to add the aspectj-plugin to the build phases. The plugin has its own dependencies. You also need to explicitely specifiy libraries containing aspects (spring-aspects and spring-data-neo4j)

```
<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>
```

19.2. Setting Up Spring Data Graph - Spring Configuration

The concrete configuration for Spring Data Graph is quite verbose as there is no autowiring involved. It sets up the following parts.

- GraphDatabaseService, IndexManager for the embedded Neo4j storage engine
- Spring transaction manager, Neo4j transaction manager
- aspects and instantiators for node and relationship entities
- EntityState and FieldAccessFactories needed for the different field handling
- Conversion services
- Finder factory
- an appropriate NodeTypeStrategy

19.2.1. XML-Namespace

To simplify the configuration we provide a xml namespace `datagraph` that allows configuration of any Spring Data Graph project with a single line of xml code. There are three possible parameters.

You can use `storeDirectory` or the reference to `graphDatabaseService` alternatively. For cross-store configuration just refer to an `entityManagerFactory`.

```
<?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>
```

```
    <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"/>
```

```
  <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>
```

19.2.2. Java based Configuration

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

> **Note**
>
> For those not familiar with how to configure the Spring container using Java based bean metadata instead of XML based metadata see the high level introduction in the reference docs here as well as the detailed documentation here.

To help configure Spring Data Graph using Java based bean metadata the class `Neo4jConfiguration` is registerd with the context either explicitly in the XML config or via classpath scanning for classes that have the @Configuration annotation. The only thing that must be provided in

addition is the `GraphDatabaseService` configured with a datastore directory. The example below shows using XML to register the `Neo4jConfiguration` @Configuration class as well as Spring's `ConfigurationClassPostProcessor` that transforms the @Configuration class to bean definitions.

```
<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 20. Cross-store persistence with a graph database

The Spring Data Graph project support cross-store persistence which allows parts of the data mode to be stored in a traditional JPA datastore (RDBMS) and other parts of the data model (even partial entites, that is some properties or relationships) in a graph store.

This allows existing JPA-based applications to embrace NOSQL data stores to evolve certain parts of their model. Possible use cases are adding social network or geospatial information to existing applications.

20.1. Partial graph persistence

Partial graph persistence is achieved by restricting the Spring Data Graph aspects to explicitly annotated parts of the entity. Those fields will be made transient by the aspect so that JPA ignores them and won't try to persist those attributes.

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

The connection between the two entities is kept 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 NodeTypeStrategy that preserves the Java type hierarchy within the graph. With 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. So it is possible on instantiation of a JPA id via the entity manager (or some other means like creating the POJO and setting its id manually) to find the matching node using the index facilities and reconnect them.

Using those mechanisms and the Spring Data Graph aspects a single POJO can contain fields that are handled by JPA and other fields (which might be relationships as well) that are handled by Spring Data Graph.

20.1.1. @NodeEntity(partial = "true")

When annotating an entity with partial true, Spring Data Graph assumes that this is a cross-store entity. So its only responsibility is for the fields annotated with Spring Data Graph annotations. JPA should not take care of these fields (they should be annotated with @Transient). In this mode of operation Spring Data Graph also handles the cross-store connection via the content of the JPA id field.

20.1.2. @GraphProperty

For common fields containing primitive or convertible values that wouldn't have to be annotated in exclusive Spring Data Graph operations this explicit declaration is necessary to be sure that they are intended to be stored in the graph. These fields should then be made transient so that JPA doesn't try to take care of them as well.

The following example is taken from the Spring Data Graph examples, it is contained in the myrestaurant-social project.

```
@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;
    }
}
```

20.2. Configuring cross-store persistence

Configuring cross-store persistence is done similarly to the default Spring Data Graph operations. As soon as you refer to an `entityManagerFactory` in the xml-namespace it is set up for cross-store persistence.

```
<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 21. Samples

21.1. Introduction

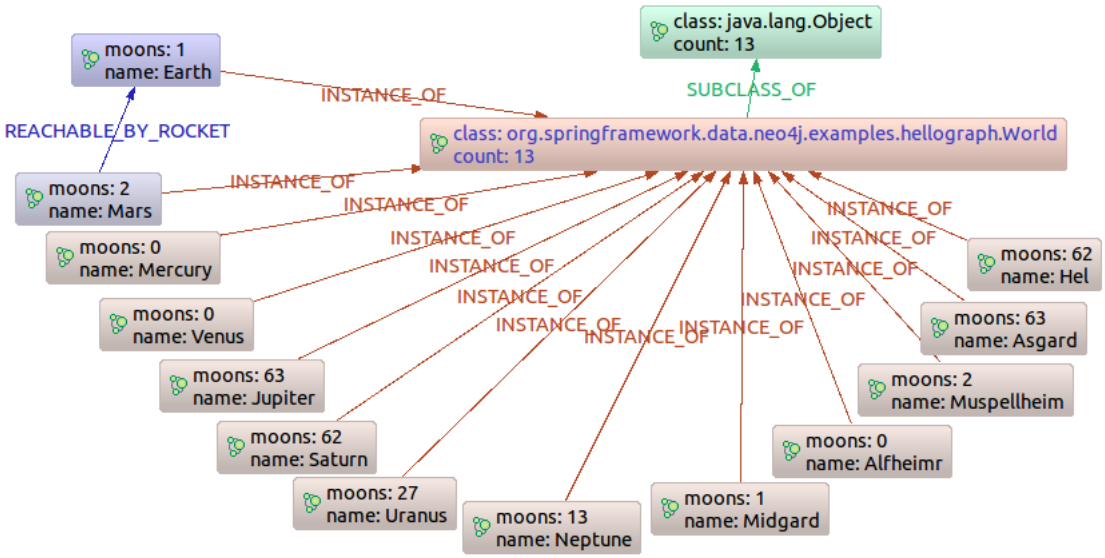
Spring Data Graph comes with a number of samples. The source code of the samples is found on GitHub. The different sample projects are introduced below.

21.2. Hello Worlds sample

The Hello Worlds sample application is a simple console application with unit tests, that creates some Worlds (entities / nodes) and Rocket Routes (relationships) in a Galaxy (graph) and then reads them back and prints them out.

The unit tests demonstrate some other features of Spring Data Graph. 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:

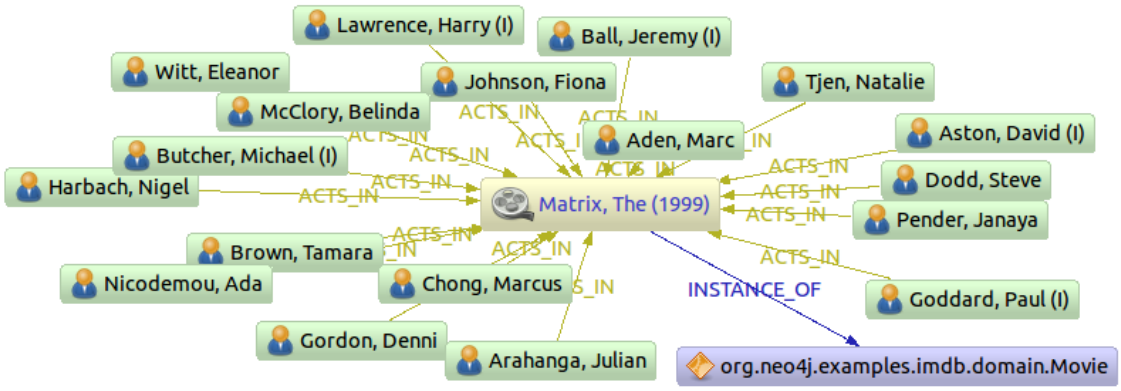


21.3. IMDB sample

A web application that imports datasets from the Internet Movie Database (IMDB) into the graph database. It allows listings of movies with their actors and actors with their roles in different movies. It also uses graph traversal operations to calculate the Kevin Bacon number (distance to an actor that has acted with Kevin Bacon). This sample application shows the basic usage of Spring Data Graph in a more complex setting with several annotated entities and relationships as well as usage of indices and graph traversal.

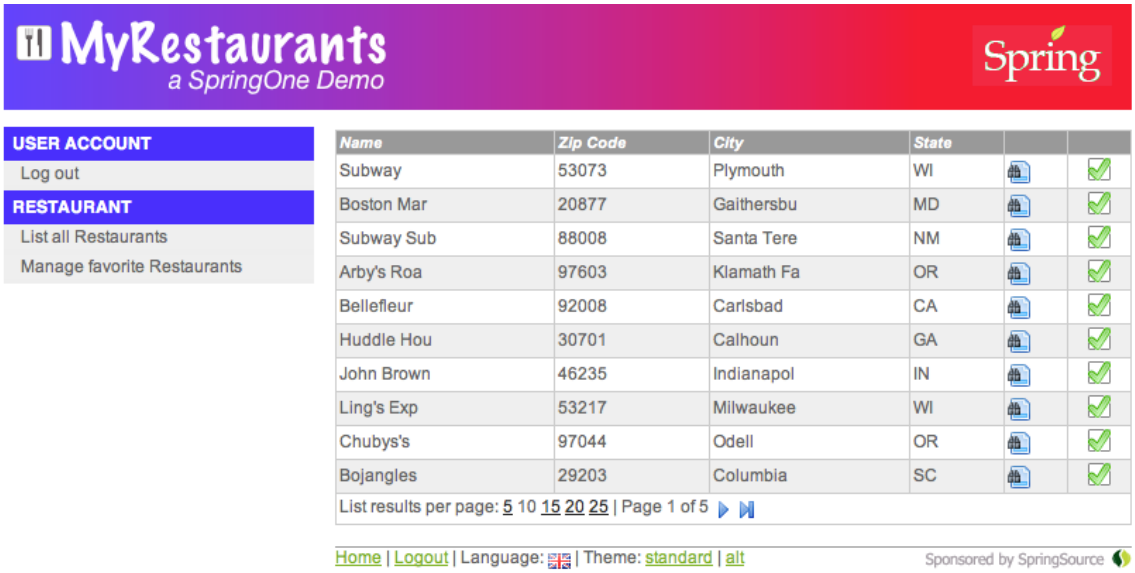
See the readme file for instruction on how to compile and run the application.

An excerpt of the data stored in the Graph Database after executing the application:



21.4. MyRestaurant sample

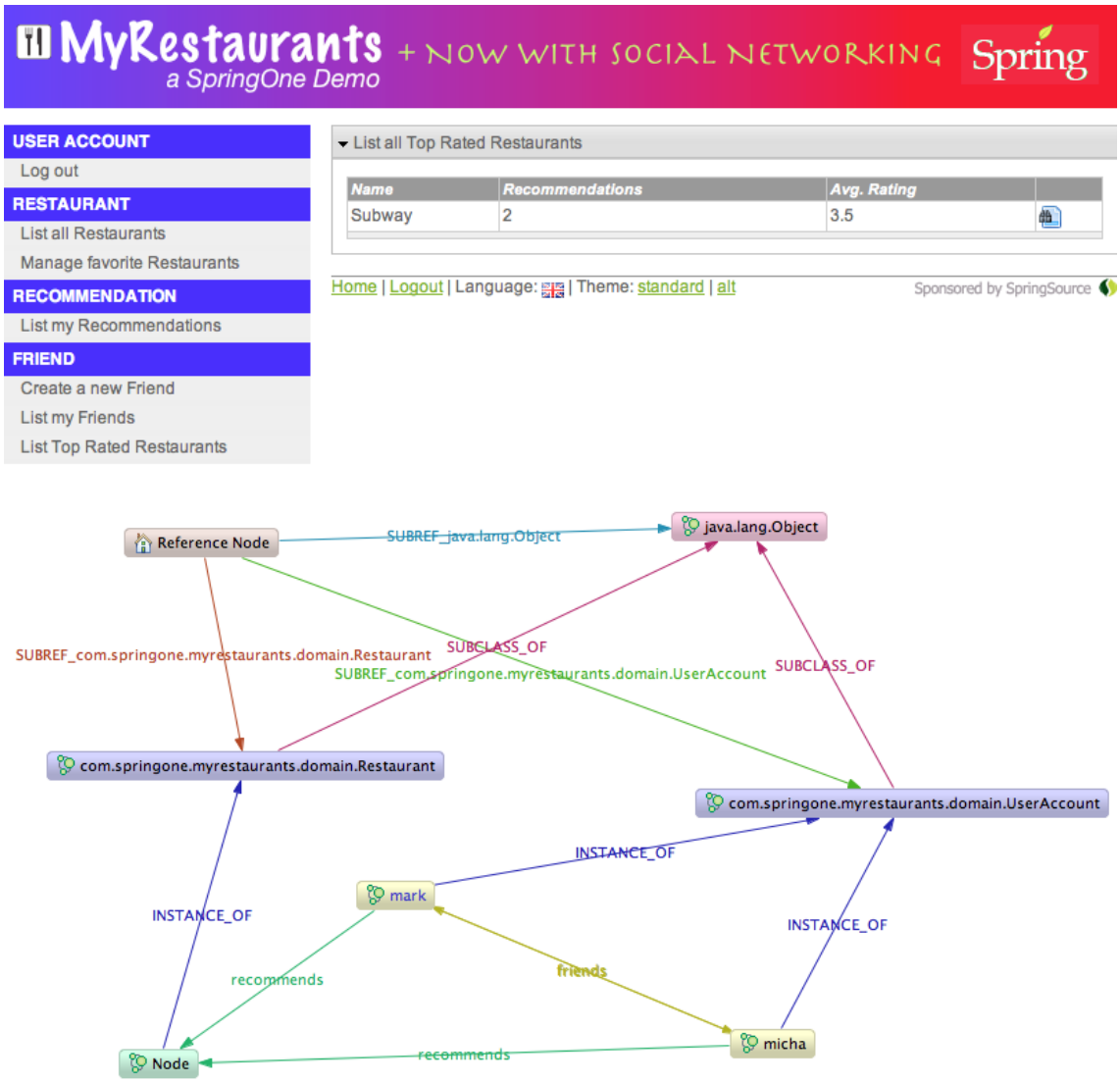
Simple, JPA based web application for managing users and restaurants, with the ability to add restaurants as favorites to a user.



21.5. MyRestaurant-Social sample

An extended version of the MyRestaurant sample application that adds social networking functionality to it. It is possible to have friends and to add rated relationships to restaurants. The relationships and some of the properties of the entities are transparently stored in the graph database. There is also a graph traversal that provides a recommendation based on your friends' (and their friends') rating of restaurants.

An excerpt of the data stored in the Graph Database after executing the application:


MyRestaurants
a SpringOne Demo
+ NOW WITH SOCIAL NETWORKING
Spring
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
Node
recommends
micha

Chapter 22. Performance considerations

Although adding another layer of abstraction is always the solution to look for in software development, each of those layers adds overhead and performance penalties. This chapter discusses the performance implications of using Spring Data Graph on top of the native Neo4j API.

22.1. When to use SDG?

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

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

Spring Data Graph was not designed with a major performance focus. It adds some overhead to pure graph operations. Something to keep in mind is, that the access of properties and relationships is a read trough in the attached case. So to avoid multiple read-throughs it is sensible to store the result in a local variable at the scope of use (method, class or jsp for example).

Most of the overhead comes from the use of the Java Reflection API, which is leveraged 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 23. Neo4jTemplate

...

23.1. Transaction handling/management

...

23.2. Basic operations

...

23.3. Indexing

...

23.4. Traversal

...

23.5. PathMapper

... path as general return type?

Chapter 24. Annotation-driven persistence

...

24.1. Annotations

... this is the most important part

24.2. Introduced methods

...

24.3. Finders

...

24.4. GraphDatabaseContext

...

24.5. Indexing

...

24.6. Traversal

...

24.7. EntityMapper and Path

...

Chapter 25. AspectJ introduction

The object graph mapper of Spring Data Graph relies heavily on AspectJ. AspectJ is the 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 repetitive tasks in a system (e.g. logging, security, auditing, caching, transaction scoping) that are difficult to extract using the normal OO paradigms. The means of the OO paradigm, of subclassing, polymorphism, overriding and delegation are still very cumbersome to use with many of those concerns applied in the codebase. Also the flexibility is limited or would add quite a number of configuration options or parameters.

The learning curve for the AspectJ pointcut language is quite slow but the developer who uses Spring Data Graph will not be confronted with that. Users do not have care about to hooking into a framework mechanism or having to extend a framework superclass.

That's why AspectJ uses a declarative approach, defining concrete advice, which is just the piece of code that contains the implementation of the concern. AspectJ advice can for instance be applied before, after, or instead of a method or constructor call, or variable access. This is declared using AspectJ's expressive pointcut language that is able to express any place within a code structure or flow. AspectJ is also able to introduce new methods, fields, annotations, interfaces, and superclasses to existing classes.

Spring Data Graph uses both mechanisms internally. First, when encountering `@NodeEntity` or `@RelationshipEntity` annotations it introduces a new interface `NodeBacked` or `RelationshipBacked`, depending on the annotation type. Secondly, it introduces fields and methods to the annotated class. See Section 18.8, “Methods added to entity classes” 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? This can be either done at compile time with the AspectJ Java compiler (ajc) that takes source files and aspect definitions, and then 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 bytecode of the classes has already been rewritten to delegate 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, for example load-time weaving and runtime weaving. These are currently not supported by Spring Data Graph.

Chapter 26. Neo4j introduction

...

Chapter 27. Spring Data

...

Chapter 28. Neo4j Server

...