Foreword by Rod Johnson

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

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

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

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

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

Enjoy the Spring Data Neo4j book, and happy coding!

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

Foreword by Emil Eifrem

"Spring is the most popular middleware on the planet," I thought to myself as I walked up to Rod Johnson in late 2009 at the JAOO conference in Aarhus, Denmark. Rod had just given an introductory presentation about Spring Roo and when he was done I told him "Great talk. You're clearly building a stack for the future. What about support for non-relational databases?"

We started talking and quickly agreed that NOSQL will play an important role in emerging stacks. Now, a year and half later, Spring Data Neo4j is available in its first stable release and I'm blown away by the result. Never before in any environment, in any programming framework, in any stack, has it been so easy and intuitive to tap into the power of a graph database like Neo4j. It's a testament to the efforts by an awesome team of four hackers from Neo Technology and VMware: Michael Hunger, David Montag, Thomas Risberg and Mark Pollack.

The Spring framework revolutionized how we all wrote enterprise Java applications and today it's used by millions of enterprise developers. Graph databases also stand out in the NOSQL crowd when it comes to enterprise adoption. You can find graph databases used in areas as diverse as network management, fraud detection, cloud management, anything with social data, geo and location services, master data management, bioinformatics, configuration databases, and much more.

Spring developers deserve access to the best tools available to solve their problem. Sometimes that's a relational database accessed through JPA. But more often than not, a graph database like Neo4j is the perfect fit for your project. I hope that Spring Data Neo4j will give you access to the power and flexibility of graph databases while retaining the familiar productivity and convenience of the Spring framework.

Enjoy the Spring Data Neo4j guide book and welcome to the wonderful world of graph databases!

Emil Eifrem, CEO of Neo Technology

Part I. Tutorial

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

The tutorial takes the reader through the steps necessary to create the application. It provides the configuration and code examples that are needed to understand what's happening in Spring Data Neo4j. The complete source code for the app is available on Github [http://spring.neo4j.org/cineasts].

Chapter 1. Introducing our project

Allow me to introduce Cineasts.net

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

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

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

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

Chapter 2. The Spring stack

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

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

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

2.1. Required setup

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

Example 2.1. Project pom.xml

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

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

Example 2.2. Project web.xml

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

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

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

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

Example 2.3. applicationContext.xml

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

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

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

Example 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:prefix="/WEB-INF/views/" p:suffix=".jsp"/>
```

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

Chapter 3. The domain model

Setting the stage

We wanted to outline the domain model before diving into library details. We also looked at the data model of the themoviedb.org data to confirm that it matched our expectations.
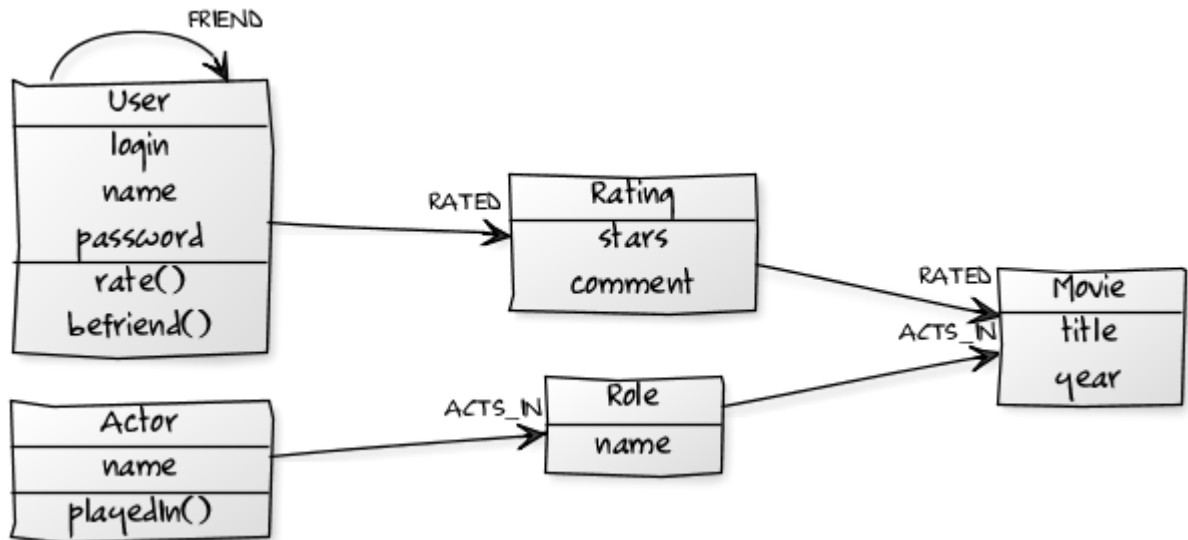
In Java code this looks pretty straightforward:

Example 3.1. Domain model

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

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

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

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

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

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

Chapter 4. Learning Neo4j

Graphs ahead

Now we needed to figure out how to store our chosen domain model in the chosen database. First we read up about graph databases, in particular our chosen one, Neo4j [http://neo4j.org]. The Neo4j data model consists of nodes and relationships, both of which can have key/value-style properties. What does that mean, exactly? Nodes are the graph database name for records, with property keys instead of column names. That's normal enough. Relationships are the special part. In Neo4j, relationships are first-class citizens, meaning they are more than a simple foreign-key reference to another record, relationships carry information. So we can link together nodes into semantically rich networks. This really appealed to us. Then we found that we were also able to index nodes and relationships [http://docs.neo4j.org/chunked/milestone/indexing.html] by {key, value} pairs. We also found that we could traverse relationships both imperatively using the core API, and declaratively using a query-like Traversal Description [http://docs.neo4j.org/chunked/milestone/tutorials-java-embedded-traversal.html]. Besides those programmatic traversals there was the powerful graph query language called Cypher [http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html] and an interesting looking DSL named Gremlin [https://github.com/tinkerpop/gremlin/wiki]. So lots of ways of working with the graph.

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

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

Example 4.1. Neo4j Maven dependency

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>1.8</version>
</dependency>
```

Example 4.2. Neo4j core API (transaction code omitted)

```
enum RelationshipTypes implements RelationshipType { ACTS_IN };

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

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

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

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

Chapter 5. Spring Data Neo4j

Conjuring magic

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

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

The first step was to configure Maven:

Example 5.1. Spring Data Neo4j Maven configuration

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

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

Example 5.2. Spring Data Neo4j context configuration

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

Chapter 6. Annotating the domain

Decorations

Looking at the Spring Data Neo4j documentation, we found a simple Hello World example [http://spring.neo4j.org/helloworld] and tried to understand it. We also spotted a compact reference card [http://spring.neo4j.org/notes] which helped us a lot. The entity classes were annotated with `@NodeEntity`. That was simple, so we added the annotation to our domain classes too. Entity classes representing relationships were instead annotated with `@RelationshipEntity`. Property fields were taken care of automatically. The only additional field we had to provide for all entities was an id-field to store the node- and relationship-ids.

Example 6.1. Movie class with annotation

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

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

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

Example 6.2. First test case

```
@Autowired Neo4jTemplate template;

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

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

Chapter 7. Indexing

Do I know you?

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

Example 7.1. Exact Indexing for Movie id

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

@Autowired Neo4jTemplate template;

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

Chapter 8. Repositories

Serving a good cause

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

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

Example 8.1. Movie repository

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

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

Example 8.2. Repository context configuration

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

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

Example 8.3. Usage of a repository

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

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

Example 8.4. Derived movie-repository query method

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

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

Example 8.5. Annotated movie-repository query method

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

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

Chapter 9. Relationships

A convincing act

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

9.1. Creating relationships

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

Example 9.1. Role class

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

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

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

Example 9.2. Relating actors to movies

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

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

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

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

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

9.2. Accessing related entities

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

Example 9.3. @RelatedTo usage

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

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

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

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

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

9.3. Accessing the relationship entities

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

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

Example 9.4. @RelatedToVia usage

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

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

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

Chapter 10. Get it running

Curtains up!

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

10.1. Populating the database

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

Example 10.1. Populating the database - Controller

```
@Service
public class DatabasePopulator {

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

@Controller
public class MovieController {

    @Autowired private DatabasePopulator populator;

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

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

10.2. Inspecting the datastore

Being the geeks we are, we also wanted to inspect the raw data in the database. Reading the Neo4j docs [http://docs.neo4j.org/], there were a couple of different ways of going about this.

10.2.1. Neoclipse visualization

First we tried Neoclipse, an Eclipse RCP application that opens an existing graph store and visualizes its content. After getting an exception about concurrent access, we learned that we have to use Neoclipse in read-only mode when our webapp was still running. Good to know.
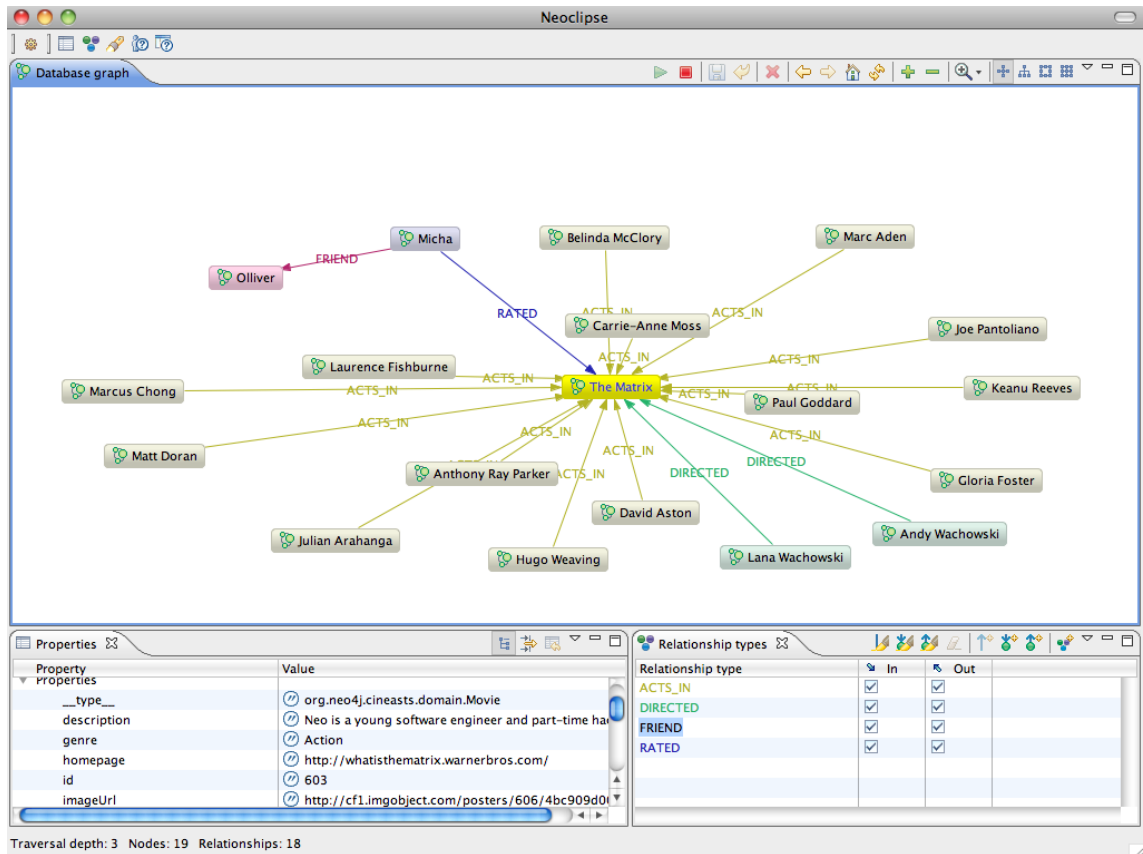
10.2.2. The Neo4j Shell

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

Example 10.2. Starting the Neo4j Shell

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

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

Example 10.3. Neo4j Shell usage

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


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


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


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


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


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

Chapter 11. Web views

Showing off

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

Example 11.1. Controller for showing movies

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

Example 11.2. Populating the database - JSP /movies/show

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

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

The UI had now evolved to this:
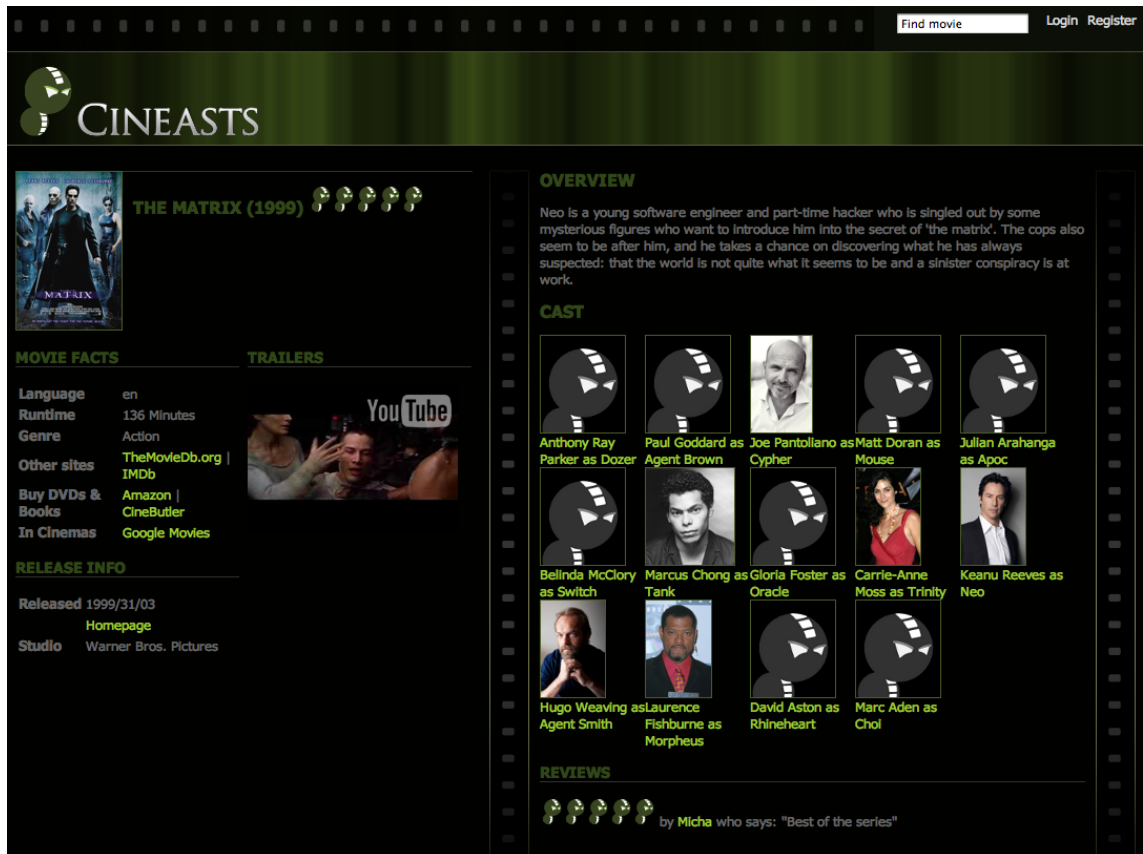
11.1. Searching

The next thing was to allow users to search for movies, so we needed some fulltext search capabilities. As the default index provider implementation of Neo4j is based on Apache Lucene [http://lucene.apache.org/java/docs/index.html], we were delighted to see that fulltext indexes were supported out of the box.

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

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

Example 11.3. Searching for movies

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

11.2. Listing results

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

Example 11.4. Search controller

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

Example 11.5. Search Results JSP

```
<h2>Movies</h2>

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

The UI now looked like this:

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. Adding social

Movies 2.0

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

12.1. Users

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

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

Example 12.1. Social entities

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

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

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

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

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

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

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

Example 12.2. Populate users and ratings

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

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

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

12.2. Ratings for movies

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

Example 12.3. Getting the rating of a movie

```
class Movie {
    ...

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

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

Fortunately our tests highlighted the division by zero error when calculating the stars for a movie without ratings. The next steps were to add this information to the movie presentation in the UI, and creating a user profile page. But for that to happen, users must first be able to log in.

Chapter 13. Adding Security

Protecting assets

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

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

Example 13.1. Spring Security pom.xml

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

Example 13.2. Spring Security web.xml

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

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

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

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

Example 13.3. Spring Security applicationContext-security.xml

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

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

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

Example 13.4. CinceastUserDetailsService interface and UserRepository custom implementation

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

    User getUserFromSession();

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

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

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

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

    User findByLogin(String login);
}

public class UserRepositoryImpl implements CineastsUserDetailsService {

  @Autowired private Neo4jOperations template;

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

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

  @Override
  public User getUserFromSession() {
      SecurityContext context = SecurityContextHolder.getContext();
      Authentication authentication = context.getAuthentication();
      Object principal = authentication.getPrincipal();
      if (principal instanceof CineastsUserDetails) {
          CineastsUserDetails userDetails = (CineastsUserDetails) principal;
          return userDetails.getUser();
      }
      return null;
  }
}

public class CineastsUserDetails implements UserDetails {
    private final User user;

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

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

Any logged-in user was now available in the session, and could be used for all the social interactions. The remaining work for this was mainly adding controller methods and JSPs for the views. We used the helper method `getUserFromSession()` in the controllers to access the logged-in user and put it in the model for rendering. Here's what the UI had evolved to:
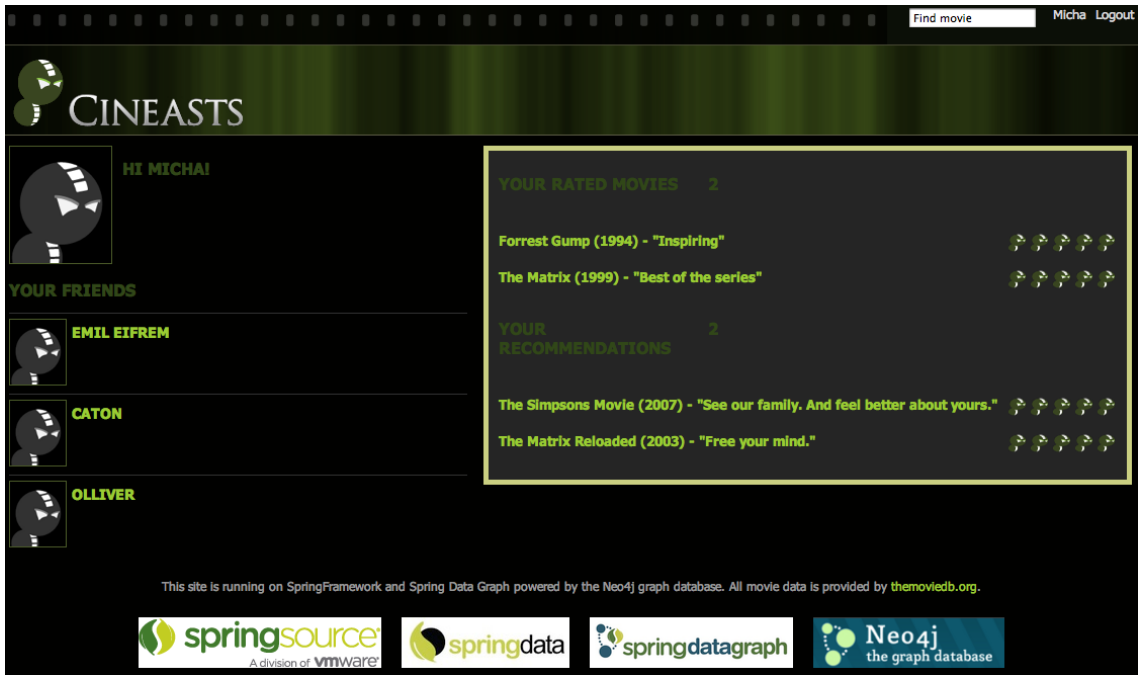
Chapter 14. More UI

Oh the glamour

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

Find movie   Login Register

CINEASTS

THE MATRIX (1999)

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

MOVIE FACTS

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

TRAILERS
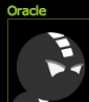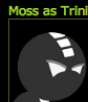
RELEASE INFO

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

REVIEWS

by Micha who says: "Best of the series"

Find movie   Micha Logout

CINEASTS

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

KEANU REEVES

Born 1964/02/09 in Beirut, Lebanon.

Chapter 15. Importing Data

The dusty archives

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

Example 15.1. JSON movie response

```
[{"popularity":3,
"translated":true, "adult":false, "language":"en",
"original_name":"[Rec]", "name":"[Rec]", "alternative_name":"[REC]",
"movie_type":"movie",
"id":8329, "imdb_id":"tt1038988", "url":"http://www.themoviedb.org/movie/8329",
"votes":11, "rating":7.2,
"status":"Released",
"tagline":"One Witness. One Camera",
"certification":"R",
"overview":"\"REC\" turns on a young TV reporter and her cameraman who cover the night shift
 at the local fire station...
"keywords":["terror", "lebende leichen", "obsession", "camcorder", "firemen", "reality tv ",
 "bite", "cinematographer",
"attempt to escape", "virus", "lodger", "live-reportage", "schwerverletzt"],
"released":"2007-08-29",
"runtime":78,
"budget":0,
"revenue":0,
"homepage":"http://www.3l-filmverleih.de/rec",
"trailer":"http://www.youtube.com/watch?v=YQUkX_XowqI",
"genres":[{"type":"genre",
"url":"http://themoviedb.org/genre/horror",
"name":"Horror",
"id":27}],
"studios":[{"url":"http://www.themoviedb.org/company/2270", "name":"Filmax Group", "id":2270}],
"languages_spoken":[{"code":"es", "name":"Spanish", "native_name":"Espa\u00f1ol"}],
"countries":[{"code":"ES", "name":"Spain", "url":"http://www.themoviedb.org/country/es"}],
"posters":[{"image":{"type":"poster",
"size":"original", "height":1000, "width":706,
"url":"http://cf1.imgobject.com/posters/3a0/4cc8df415e73d650240003a0/rec-original.jpg",
"id":"4cc8df415e73d650240003a0"}},
....
"cast":[{"name":"Manuela Velasco",
"job":"Actor", "department":"Actors",
"character":"Angela Vidal",
"id":34793, "order":0, "cast_id":1,
"url":"http://www.themoviedb.org/person/34793",
"profile":"http://cf1.imgobject.com/profiles/390/.../manuela-velasco-thumb.jpg"},
...
{"name":"Gl\u00f2ria Viguer",
"job":"Costume Design", "department":"Costume \u0026 Make-Up",
"character":"",
"id":54531, "order":0, "cast_id":21,
"url":"http://www.themoviedb.org/person/54531",
"profile":""}],
"version":150, "last_modified_at":"2011-02-20 23:16:57"}]
```

Example 15.2. JSON actor response

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

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

Example 15.3. Importing the data

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

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

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

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

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

Chapter 16. Recommendations

Movies! Friends! Bargains!

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

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

Example 16.1. Cypher based movie recommendation on Repository

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

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

Example 16.2. Cypher - Friend Recommendation on Repository

```
interface UserRepository extends GraphRepository<User> {
 @Query("
 start user=node({0})
 match user-[r:RATED]->movie<-[r2:RATED]-likeminded,
   user-[:FRIEND]-friend
 where r.stars > 3 and r2.stars >= 3
 return likeminded
 order by count(*) desc
 limit 10
 ")
    Iterable<User> suggestFriends(User me);
}
```

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

Chapter 17. Conclusion

To new frontiers

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

But we leave you with that as an exercise to enjoy and explore. Thanks for following the tutorial and make sure to get back to us with suggestions for improvements or reports about unexpected behaviours at the discussion forums [http://spring.neo4j.org/discussions], or the issue tracker [http://spring.neo4j.org/issues].