Spring Data Graph - Reference Documentation

1.0.0.M1

Part I. Introduction

This document is the reference guide for Spring Data - Graph Support. It explains Spring Data Graph concepts, usage, infrastructure of the framework and underlying semantics for the underlying graph stores.

For an introduction to graph stores or Spring, or Spring Data examples, please refer to Chapter 3, *Getting Started* - this documentation refers only to Spring Data Graph Support and assumes the reader is familiar with Spring concepts.

Chapter 1. Why Spring Data Graph?

NOSQL stores provide alternative storage solutions that are more tailored to the needs of the data structure requirements that are specific to each project than just using a relational database as a "one-size-fits-all" solution.

Graph databases provide excellent support for network data, i.e. data that easily can be structured as connected nodes. Property graph databases like Neo4j support arbitrary numbers of named properties on both nodes and relationships. They are highly performant when traversing large, complex datasets with millions of nodes and relationships, even on commodity hardware.

Neo4j is an open source graph database written in Java. It has excellent performance characteristics while providing ACID semantics and transactional support (both JTA and XA transactions). Neo4j can run as a lightweight embedded database as well as a standalone server that exposes the API via a rich REST interface.

The Spring Data Graph (or DATAGRAPH) framework makes it easy to integrate graph databases in existing or new Spring applications by providing infrastructure that reduces the amount of boilerplate data access code and uses commonly known patterns and idioms that are well known in the Spring Framework community. Those practices are based on a simple POJO programming model that leverages annotations to add metadata and can be integrated in any part of a spring application, like the web or service layers.

A special use case of Spring Data Graph is the cross store solution that can extend existing JPA data models with new parts (properties, entities, relationships) that are stored exclusively in the graph while being transparently integrated with the JPA entities. This enables for easy and seamless addition of new features that were not available before to JPA-based applications.

Chapter 2. Requirements

Spring Data Graph 1.x binaries requires JDK level 6.0 and above, and Spring Framework 3.0.x and above.

In terms of graph databases, Neo4j version 1.2 and above. Neo4j has a dependency on Apache Lucene for indexing. Users are encouraged to use the latest version of Neo4j available.

For building the project, Apache Maven (version 2.10 and above) is strongly recommended.

Chapter 3. Getting Started

The focus on NOSQL databases is a recent one, even if many of those stores have existed for some years now. That's why this document will not only guide you through the relevant parts of the DATAGRAPH API, but also explain some key concepts of graph databases.

After reading this document, you should be able to integrate DATAGRAPH in your own existing or new applications. If there are any issues that you don't understand or think are explained in a too complicated way, please report back any problems or suggestions. This would also benefit future readers of this documentation.

3.1. First Steps

As explained in Chapter 1, *Why Spring Data Graph?*, Spring Data Graph (DATAGRAPH) provides integration between the Spring framework and graph databases. Familiarity with the Spring framework is assumed as stated in Part I, “Introduction”, and only minimally cross referenced here. Graph databases and Neo4j in particular are explained in a bit more detail. The main focus of this document is however on explaining the steps needed to get a DATAGRAPH-backed application up and running.

3.1.1. Knowing Spring

DATAGRAPH makes heavy use of Spring Framework's core functionality, such as the IoC container, converter API and the AOP infrastructure. While it is less important to know the Spring APIs, understanding the concepts behind them is. The Spring Framework documentation home page is a good starting point for developers wanting become more familiar with Spring Framework.

3.1.2. Knowing NOSQL and graph databases

The recent interest in NOSQL databases is mainly driven by the need for finding the best suited storage solution for data structured in a specific way. It should fit the data, not the other way round. Another issue is the scalability of the database, especially with today's fast growing user bases handling large amounts of data in a short time. There are many NOSQL databases, and one should become familiar with the different concepts, advantages, and disadvantages before choosing a solution. A problem with the NOSQL databases is the different data access APIs that are provided. Spring Data aims at easing this burden by providing consistent abstractions over those APIs, leveraging SpringSource's experience and good reputation in this area.

Graph databases are a particularly good fit for large networks of connected information (objects). They map objects to nodes and connections to relationships. Examples of such datasets are social networks, geospatial information, network layouts, and hardware or dependency graphs. Neo4j is the first graph database that is tightly integrated with the DATAGRAPH project.

3.1.3. Trying Out The Samples

DATAGRAPH comes with a number of samples and unit test cases (if you accessed the sources via github or maven).

The current distribution contains:

- 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 DATAGRAPH. The sample comes with a minimal config for maven and spring to get up and running quickly.

- 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 a actor that has acted with Kevin Bacon). This sample application shows the basic usage of DATAGRAPH in a more complex setting with several annotated entities and relationships as well as usage of indices and graph traversal.

- MyRestaurant sample

  Simple webapp for managing users and restaurants, with the ability to add a restaurants as favorites to a user.

- 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.

Most of the samples are web applications that can be easily built and run using mvn jetty:run.

3.2. Need Help?

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

3.2.1. Spring Data homepage

The Spring Data homepage provides all the necessary links for information, community forums and code repositories.

3.2.2. Professional Support

Professional, from-the-source support, with guaranteed response time, is available from SpringSource, the company behind Spring Data and Spring.

3.3. Following Development

For information on the Spring Data source code repository, nightly builds and snapshot artifacts please see the Spring Data home page.

You can help make Spring Data best serve the needs of the Spring community by interacting with developers through the community forums.

If you encounter a bug or want to suggest an improvement, please create a ticket on the DATAGRAPH issue tracker.

To stay up to date with the latest news and announcements in the Spring eco system, subscribe to the Spring Community Portal.

Lastly, you can follow the SpringSource Data blog or the project team on Twitter (@SpringData)

Part II. Reference

This part of the reference documentation details the API, concepts, annotations, datastore, programming model and the cross-store approach 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 DATAGRAPH 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, relationships 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 DATAGRAPH backed application only the public annotations are relevant, basic knowledge of graph stores is needed to access advanced functionality like traversals. Traversals results can also be mapped to fields of entities.

Chapter 4. Introduction to the Neo4j Graph Database

Neo4j is a graph database, a fully transactional database that stores data structured as graphs. A graph is a flexible data structure that allows for a more agile and rapid style of development.

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 provides:

- an intuitive graph-oriented model for data representation. Instead of static and rigid tables, rows and columns, you work with a flexible graph network consisting of nodes, relationships and properties.
- a disk-based, native storage manager completely optimized for storing graph structures for maximum performance and scalability.
- massive scalability. Neo4j can handle graphs of several billion nodes/relationships/properties on a single machine and can be sharded to scale out across multiple machines.
- a powerful traversal framework for high-speed traversals in the node space.
- can be deployed as a full server or a very slim database with a small footprint (~500k jar).
- a simple and convenient object-oriented 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-strength database. Neo4j is released under a dual free software/commercial license model.

4.1. 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 properties to nodes and relationships which can be used directly or during traversals.

4.2. GraphDatabaseService

The GraphDatabaseService is the API interface to the storage engine. It provides access to create and retrieve Nodes and Relationships, an IndexManager, lifecycle and transactional methods and more.

The EmbeddedGraphDatabaseService is running within the current Java application for highest performance and tightest integration. There are other, remote implementations that provide access to Neo4j stores via REST or RMI.

4.3. 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 relationship can have properties. Property values can be of primitive Java types and

Strings as well as byte arrays for binary data. Node creation and modification has to happen within a transaction, reading from the graph store can also be done without transactions.

```
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();
}
```

4.4. Graph traversal

Getting a single nodes or relationship and reading from 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 );
}
```

Chapter 5. 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 DATAGRAPH and how to use them. Examples for this section are taken from the imdb project of DATAGRAPH examples.

5.1. Overview of the AspectJ support

Behind the scenes DATAGRAPH leverages AspectJ aspects to modify the behavior of simple POJO entities to be able to be backed by a graph store. Behind each entity lies a backing node that holds the 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 `getUnderlyingState()` 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 EntityStateAccessors 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 EntityStateAccessors then use a number of FieldAccessor factories to create a FieldAccessor instance per field that does the specific handling needed for the concrete field.

5.2. Using annotations to define POJO entities and relationships

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

5.2.1. Entities with @NodeEntity

This annotation is used to declare a POJO entity to be backed by a node in the graph store. Its field are mapped by default to either properties of the node. Fields pointing to other NodeEntities or Collections thereof are mapped via relationships. If the annotation field `useShortNames` is set to false the properties and relationship names used will be prepended with the class name of the entity. If the field `fullIndex` is set to true, all fields of the entity will be indexed. If the `partial` field is set to true, this entity takes part in a cross-store setting where only the parts of the entity are mapped to the graph store that are not handled by JPA.

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

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

5.2.2. RelationshipEntities with @RelationshipEntity

To access the rich datamodel of graph relationship 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 (and are read only).

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

5.2.3. Fields with @GraphProperty

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 servcies will stored as a string. Transient fields are not persisted. This annotation is mainly used for cross store persistence. Future uses can be specification of property names and such.

5.2.4. Fields with @RelatedTo pointing to other 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 the simple relationships to other node entities don't have to be annotated as DATAGRAPH can extract all needed information from the field using reflection. In the case of multi-relationships at least the target node entity java class defined in the `elementClass` field of the @RelatedTo annotation is needed. The `direction` (default OUTGOING) and `type` (inferred from field name) attributes of the annotation are optional.

Relationships to single node entities are created on setting the field and deleted on 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;
}
```

5.2.5. Fields with @RelatedToVia pointing to 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;
}
```

5.2.6. @StartNode

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

5.2.7. @EndNode

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

5.2.8. @Indexed

The @Indexed annotation can be declared on fields that are intended to be indexed on modification by the Neo4j IndexManager. The resulting index can be used to later retrieve nodes or sets of nodes or relationships that contain a certain property value, e.g. as start node for a traversal. The index access is available via a Finder instance created by a FinderFactory for a given node or relationship entity class. DATAGRAPH supports only the _new_ index API introduced with Neo4j 1.2. GraphDatabaseContext exposes the Indexes for Nodes and Relationships. Indexes can be named, for instance to keep separate domain concepts in separate indexes. Thats 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 for all fields of the entity. If no index name is specified, they default to the ones configured with neo4j ("node" and "relationship").

5.2.9. @GraphTraversal

The @GraphTraversal annotation leverages the delegation infrastructure used by the DATAGRAPH aspects. It provides dynamic fields which on access 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.

5.3. Finding Nodes with Finders

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

- using direct access (findById),
- 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 instances with a certain property value (findByPropertyValue),
- iterating over a traversal result (findAllByTraversal),

The Finder instances are created via a FinderFactory to be bound to a concrete node entity class.

5.4. Representing Java Types via NodeTypeStrategy

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. That's why the hierarchy up to `java.lang.Object` of all these classes will be persisted in the graph. 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.

The current implementation uses nodes to represent the java type hierarchy which are connected via SUBCLASS_OF relationships to their superclass nodes and via INSTANCE_OF relationships to the concrete node entity instance node.

An alternative approach could use indexing operations to perform the same functionality.

5.5. 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. Neo4j also provides a Spring compliant transaction manager that allows it to participate in Spring managed transactions (also with @Transactional). This transaction manager is already configured in the Spring Java config, class AbstractNeo4jConfiguration.

DATAGRAPH is designed to work within transaction boundaries. So entity creation and modification should happen within transactional methods. Due to the usage of POJO entities it is common to create and populate them also outside of a transaction (e.g. in the web layer). That's why some housekeeping support was added to DATAGRAPH. It is possible to create node entities outside of transactions and also to modify their fields. Those values are then not stored within the backing node but instead only in the entity itself. When the entity reenters a transactional context and its fields are read or written to, all the pending changes are flushed to the backing node first.

By now there is no support for the creation of relationships outside of transactions and also more complex operations like creating whole subgraphs is not supported.

Chapter 6. Setup required for Spring Data Graph

To use DATAGRAPH 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 DATAGRAPH. Examples for this setup can be found in the DATAGRAPH examples.

6.1. Maven Configuration

As stated in the requirements chapter, DATAGRAPH is built easiest with Apache Maven. Its main dependencies are DATAGRAPH itself, Spring Data Commons, some parts of the Spring-Framework and of course the Neo4j graph database.

6.1.1. Repositories

The milestone releases of DATAGRAPH are available from the dedicated milestone repository. Neo4J releases are available from maven central, milestone releases are available from the Neo4j repository.

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

<repository>
        <id>neo4j-public-repository</id>
        <name>Publically available Maven 2 repository for Neo4j</name>
        <url>http://m2.neo4j.org</url>
</repository>
```

6.1.2. Dependencies

The dependency on `spring-data-neo4j` should pull in the spring framework (core, context, aop, aspects, tx), aspectj, neo4j and spring data commons dependencies. If you use the already (or in different versions) 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.M1</version>
</dependency>

<dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>1.6.10.RELEASE</version>
</dependency>
```

6.1.3. AspectJ build configuration

As DATAGRAPH uses 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>1.6.10.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjtools</artifactId>
                        <version>1.6.10.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>
```

6.2. 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
- EntityStateAccessors and FieldAccessFactories needed for the different field handling
- Conversion services
- Finder factory
- an appropriate NodeTypeStrategy

That's why DATAGRAPH provides a Spring Java Config class (annotated with @Config)

`AbstractNeo4jConfiguration` that takes care of all that. The only thing that must be provided in the custom Spring config is the `GraphDatabaseService` configured with a datastore directory. This can be achieved by extending that class and implementing the `graphDatabaseService` method.

```
public class MyConfig extends AbstractNeo4jConfiguration {
    @Override
    public boolean isUsingCrossStorePersistence() {
        return false;
    }

    @Bean(destroyMethod = "shutDown")
    public GraphDatabaseService graphDatabaseService() {
        return new EmbeddedGraphDatabase("target/neo4j-db");
    }
}
```

```
<beans>
...

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

Chapter 7. 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, i.e. some properties or relationships) in the graph store.

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

7.1. Partial graph persistence

This is achieved by allowing the DATAGRAPH aspects only to handle those parts of the entities that are explicitely annotated with DATAGRAPH annotations. Those fields have to be made transient so that JPA ignores them and won't try to persist those attributes.

A backing node in the graph store is created when the entity has been assigned a JPA id. Only then the connection between the two stores can be kept. So until the entity has been persisted, its state is just kept inside the POJO 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 makes the Java type hierarchy of the datamodel accessible in the graph. So those pieces of information can be used to 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 DATAGRAPH 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 DATAGRAPH.

7.1.1. @NodeEntity(partial = "true")

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

7.1.2. @GraphProperty

For common fields containing primitive or convertible values that wouldn't have to be annotated in exclusive DATAGRAPH operations this explicit declaration is necessary to be sure that they are intended to be stored in the graph. Those 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 DATAGRAPH 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
            @Transient
            String nickname;

            @RelatedTo(type = "friends", elementClass = UserAccount.class)
            @Transient
            Set<UserAccount> friends;

            @RelatedToVia(type = "recommends", elementClass = Recommendation.class)
            @Transient
            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, DynamicRelationshipType.withName("friends"));
            }

                @Transactional
            public Recommendation rate(Restaurant restaurant, int stars, String comment) {
                Recommendation recommendation = (Recommendation) relateTo(restaurant,
                                Recommendation.class, "recommends");
                recommendation.rate(stars, comment);
                return recommendation;
            }
            public Iterable<Recommendation> getRecommendations() {
                return recommendations;
            }
        }
```

7.2. Configuring cross-store persistence

Configuring cross store persistence is done similarly to the default DATAGRAPH operations. The concise Spring Java Config configuration class already contains a method `isUsingCrossStorePersistence`that must be implemented by a concrete configuration which controls the cross store mode of DATAGRAPH.

```
public class MyRestaurantConfig extends AbstractNeo4jConfiguration {
    @Override
    public boolean isUsingCrossStorePersistence() {
        return true;
    }

    @Bean(destroyMethod = "shutDown")
    public GraphDatabaseService graphDatabaseService() {
        return new EmbeddedGraphDatabase("target/myrestaurant-social");
    }
}
```

```
<beans>
...
        <tx:annotation-driven mode="aspectj" transaction-manager="transactionManager"/>
        <bean class="com.springone.myrestaurants.config.MyRestaurantConfig"/>
    <bean class="org.springframework.context.annotation.ConfigurationClassPostProcessor"/>
...
</beans>
```