Geomajas Hibernate layer plugin

Geomajas Developers and Geosparc

Geomajas Hibernate layer plugin

by Geomajas Developers and Geosparc

1.17.0

Table of Contents

List of Examples

Chapter 1. Introduction

This plug-in represents a vector layer implementation based upon the popular Hibernate O/R mapping framework. It uses a special spatial extension of Hibernate, unsurprisingly called Hibernate Spatial. The Hibernate Spatial project has its project website at http://www.hibernatespatial.org [http://www.hibernatespatial.org/]. The spatial extensions or dialects (in Hibernate language), that Hibernate Spatial adds, allow the definition of spatial types and the execution of spatial queries in a database independent way.

At this moment, the Hibernate layer plug-in supports the following databases:

- Oracle SDO; It has been tested with Oracle 10i, but it should work equally well with Oracle 9i and Oracle 11g.
- Postgresql/PostGIS: PostGIS 1.1.6 and higher.
- Mircosoft SQL server 2008. Currently only the Geometry types are supported.
- MySQL: Works with MySQL 5.0 and higher. Note that MySQL does not completely implement the OGC Simple Feature Specification. Some functions that work for PostGIS or Oracle may fail to work for MySQL.

The real strength of this plug-in lies in the fact that it allows you to deal with geographic data in a standardized way. It abstracts away from the specific way your database supports geographic data, and provides a standardized, cross-database interface to geographic data storage and query functions.

Chapter 2. Configuration

1. Dependencies

In order to work with the Hibernate layer plug-in, you need multiple libraries to be present on your classpath: the Geomajas Hibernate plug-in, Hibernate Spatial, and drivers for the specific database that you want to use.

Lets say, for example, you want to make use of this plug-in through a PostGIS database. Then you would need to include the following dependencies:

Example 2.1. Hibernate layer dependency

```
<dependency>
    <groupId>org.geomajas.plugin</groupId>
    <artifactId>geomajas-layer-hibernate</artifactId>
  <version>1.17.0</version>
</dependency>
<dependency>
    <groupId>org.hibernatespatial</groupId>
    <artifactId>hibernate-spatial-postgis</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>org.postgis</groupId>
    <artifactId>postgis-jdbc</artifactId>
    <version>1.1.6</version>
</dependency>
<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>8.1-407.jdbc3</version>
</dependency>
```

If you use the geomajas-project-server in your dependencyManagement section, then the versions do not need to be included;

Example 2.2. Hibernate layer dependency with using geomajas-project-server in dependency management

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.geomajas.project</groupId>
            <artifactId>geomajas-project-server</artifactId>
    <version>1.17.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
```

```
        <groupId>org.geomajas.plugin</groupId>
        <artifactId>geomajas-layer-hibernate</artifactId>
    </dependency>
    <dependency>
        <groupId>org.hibernatespatial</groupId>
        <artifactId>hibernate-spatial-postgis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgis</groupId>
        <artifactId>postgis-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>
</dependencies>
```

The Geomajas Hibernate Layer uses Hibernate 3.5.2 and Hibernate Spatial 1.0 versions. If you want to work with later versions of Hibernate or Hibernate Spatial, you need to adapt the aforementioned dependencies. Hibernate Spatial is version dependent [???]on Hibernate.

Example 2.3. Hibernate 3.6 - Hibernate spatial 1.1 dependencies

```
<dependency>
  <groupId>org.geomajas.plugin</groupId>
  <artifactId>geomajas-layer-hibernate</artifactId>
  <exclusions>
    <!--add exclusion for hibernate-annotations in geomajas-layer-hibernate, as
    <exclusion>
       <groupId>org.hibernate</groupId>
       <artifactId>hibernate-annotations</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.6.10.Final</version>
</dependency>

<dependency>
     <groupId>org.hibernatespatial</groupId>
     <artifactId>hibernate-spatial</artifactId>
     <version>1.1.1</version>
</dependency>

<dependency>
     <groupId>org.hibernatespatial</groupId>
     <artifactId>hibernate-spatial-postgis</artifactId>
     <version>1.1.1</version>
</dependency>

<dependency>
     <groupId>postgresql</groupId>
     <artifactId>postgresql</artifactId>
     <version>9.1-901.jdbc4</version>
</dependency>
```

```
<dependency>
     <groupId>org.postgis</groupId>
     <artifactId>postgis-jdbc</artifactId>
     <version>1.3.3</version>
     <exclusions>
         <exclusion>
             <groupId>org.postgis</groupId>
             <artifactId>postgis-stubs</artifactId>
         </exclusion>
     </exclusions>
</dependency>

<dependency>
    <groupId>com.jolbox</groupId>
    <artifactId>bonecp</artifactId>
    <version>0.7.1-rc3</version>
</dependency>
```

2. Basic Hibernate configuration

First of all, you need to configure the basic Hibernate library. On the root of your classpath (for a Maven project this is typically `src/main/resources/`), you need a `hibernate.cfg.xml` file. This configuration file determines the dialect to use (PostGIS in the example below), and also which Java classes that will provide the mapping onto the database. In the example below, there is one Java class (`org.geomajas.server.pojo.MyPojoClass`):

Example 2.4. Example hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.dialect">org.hibernatespatial.postgis.PostgisDi
      <property name="cache.provider_class">org.hibernate.cache.HashtableCacheP
      <property name="show_sql">false</property>

      <mapping class="org.geomajas.server.pojo.MyPojoClass" />
      .....

   </session-factory>
</hibernate-configuration>
```

It is recommended to read up on the Hibernate configuration options on the official Hibernate website [http://www.hibernate.org/], to get a better grasp of all the possibilities Hibernate provides.

Tip

In normal Hibernate applications, you would also configure you database connection parameters in the hibernate.cfg.xml. Geomajas however uses the Spring framework, and so provides the option of injecting Session factories at run-time. This can be a very powerfull approach, as it allows you to set up new DAO's etc. much quicker.

So instead of configuring all connection, session and transaction parameters through the default Hibernate configuration, it is recommended to configure these through Spring.

3. Spring transaction configuration

The Spring framework provides a connection to Hibernate by means of the `org.springframework.orm.hibernate3.HibernateTransactionManager`. This allows you to make full use of the power of dependency injection in your DAOs. In order to get up-and-running, the following parameters need to be defined (somewhere in the Spring XML files):

- The data source: this specifies the connection pool type and the connection properties of the database (PostGis in this case).
- The session factory: this is Hibernate's primary singleton and used by the Hibernate layer to access the session/connection. It also points to the Hibernate configuration file.
- A tag to enable annotation-based transactional behavior, internally used by Geomajas to decide which commands need transaction support
- The platform transaction manager for Hibernate

Here is an example wherein the above parameters are configured:

Example 2.5. Hibernate transaction configuration

```
<bean id="testDataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:mem:baseball" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>

<bean id="testSessionFactory" class="org.springframework.orm.hibernate3.Loca
    <property name="dataSource" ref="testDataSource" />
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
    <property name="configurationClass" value="org.hibernate.cfg.Annotation
</bean>

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

<bean id="transactionManager" class="org.springframework.orm.hibernate3.Hib
    <property name="sessionFactory" ref="testSessionFactory" />
</bean>

<bean name="simpleDateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg type="java.lang.String" value="dd/MM/yyyy" />
</bean>
```

4. Configuring a vector layer

Creating a vector layers with the Geomajas Hibernate plug-in takes a bit more work than using the GeoTools layers, but in return it provides you with a lot more options. In short, the following actions need to be taken in order to set up a vector layer:

- You need a spatial database with one or more tables. At least one table must have a geometric column. This of course falls out of the scope of this document, but in order to correctly configure your Java pojo objects, it is necessary for you to know what your database looks like.
- Configure your Java pojo objects that provide the mapping onto the database. Note that these Java classes must be mentioned in the hibernate.cfg.cml configuration file.
- Configure a Geomajas vector layer that makes use of the Java pojo object.

In the following section, an example vector layer will be configured. This example will demonstrate a simple case, with a one-to-one mapping between a Java class and a single spatial database table.

4.1. Spatial database table

We start out by creating a simple spatial table in the database containing "areas of interest". If you have your own data, you will be creating your own database tables.

In the SQL query below, we create a table with the name "areaofinterest" and give it 4 columns: an ID, 2 text columns (title and description) and a geometry column (type Polygon).

Example 2.6. SQL for creating a PostGIS spatial table

```
CREATE TABLE areaofinterest (
    id integer NOT NULL,
    title character varying(100) NOT NULL,
    description character varying,
    geom geometry,
    CONSTRAINT enforce_dims_geom CHECK ((ndims(geom) = 2)),
    CONSTRAINT enforce_geotype_geom CHECK (((geometrytype(geom) = 'POLYGON'::te
    CONSTRAINT enforce_srid_geom CHECK ((srid(geom) = 900913))
);
```

4.2. Java O/R mapping

Assuming that you have a database with spatial data, let us now create a Java class that maps onto that database. We will continue using the "areaofinterest" database table from the previous section. A java class `org.geomajas.server.pojo.AreaOfInterest` would now look like this:

Example 2.7. AreaOfInterest class

```
@Entity
@Table(name = "areaofinterest")
public class AreaOfInterest {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, name = "title")
    private String title;

    @Column(name = "description")
    private String description;

    @Type(type = "org.hibernatespatial.GeometryUserType")
    @Column(nullable = false, name = "geom")
    private Geometry geometry;

    // Constructors, getters, and setters
    ....
```

At the top of the class, you make a reference to the actual database table; in this case "areaofinterest". In the actual Java field declarations, you refer to the database columns within the "areaofinterest" table.

Caution

The mappings in your Java class are case sensitive! Typically PostGIS will use lowercase characters while Oracle uses uppercase characters.

Also make sure you add the Java class names to the hibernate.cfg.cml configuration file.

Tip

At this point you might want to check if the O/R mapping is correct by writing a DAO and few unit tests.

The Hibernate layer uses the same way to access you class as Hibernate itself. If your Hibernate annotations are on the fields (which is recommended), then the fields will be used for reading and writing of values. If the annotations are on the getters, then the getters and setters will be used for reading and writing of values. You can changes this at both class and field level by using an AccessType annotation. Use `@AccessType("property")` to use the getters and setters and `@AccessType("field")` to directly access the field itself.

4.3. Vector layer configuration

Now that you have a spatial database table and a Java class to map it, it's time to create the actual Geomajas vector layer configuration. Following the Geomajas configuration rules, we first configure a `org.geomajas.configuration.VectorLayerInfo` object, and only than the actual layer definition.

4.3.1. Configuring the VectorLayerInfo

The configuration of the VectorLayerInfo is almost identical as with other vector layer plug-ins. The main difference is that the dataSourceName in the FeatureInfo needs the Java classname of the mapping class (i.e. `org.geomajas.server.pojo.AreaOfInterest`), and the names of the attributes, need to point to the fields in that Java class.

So for the Java class from the previous section, we would get:

Example 2.8. VectorLayerInfo for the AreaOfInterest class

```
<!-- Area Of Interest Vector layer definition -->
<bean name="aoiInfo" class="org.geomajas.configuration.VectorLayerInfo">
    <property name="layerType" value="POLYGON" />
    <property name="crs" value="EPSG:900913" />
    <property name="maxExtent">
        <bean class="org.geomajas.geometry.Bbox">
            <property name="x" value="-20026376.393709917" />
            <property name="y" value="-20026376.393709917" />
            <property name="width" value="40052752.787419834" />
            <property name="height" value="40052752.787419834" />
        </bean>
    </property>
    <property name="featureInfo" ref="aoiFeatureInfo" />
    <property name="namedStyleInfos">
        <list>
            <ref bean="aoiStyleInfo" />
        </list>
    </property>
</bean>

<!-- Feature (attributes..) definition -->
<bean name="aoiFeatureInfo" class="org.geomajas.configuration.FeatureInfo">
    <property name="dataSourceName" value="org.geomajas.server.pojo.AreaOfInter
    <property name="identifier">
        <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
            <property name="label" value="Id" />
```

```
            <property name="name" value="id" />
            <property name="type" value="LONG" />
        </bean>
    </property>
    <property name="geometryType">
        <bean class="org.geomajas.configuration.GeometryAttributeInfo">
            <property name="name" value="geometry" />
            <property name="editable" value="false" />
        </bean>
    </property>
    <property name="attributes">
        <list>
            <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
                <property name="label" value="Title" />
                <property name="name" value="title" />
                <property name="editable" value="true" />
                <property name="identifying" value="true" />
                <property name="type" value="STRING" />
            </bean>
            <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
                <property name="label" value="Description" />
                <property name="name" value="description" />
                <property name="editable" value="true" />
                <property name="identifying" value="false" />
                <property name="type" value="STRING" />
            </bean>
        </list>
    </property>
</bean>

<!-- Style definition -->
<bean class="org.geomajas.configuration.NamedStyleInfo" name="aoiStyleInfo">
    ............Not important here.......
</bean>
```

Make sure to keep an eye on the following:

- The CRS must be the same as defined in the database table.
- The geometry type must be the same as defined in the database table.
- The dataSourceName must point to the Java class name.
- All attribute names must point to the fields in the Java class.
- Don't just copy/paste; there is no styling information in the example above ;-)

4.3.2. Configuring the layer

When the `org.geomajas.configuration.VectorLayerInfo` definition has been defined, it is time to define the actual layer. This layer must of course point to the VectorLayerInfo object that we just defined, but it must also make use of the SessionFactory that was configured:

Example 2.9. Hibernate layer definition

```
<!-- Needed when the Hibernate pojo classes contain dates. -->
<bean name="simpleDateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg type="java.lang.String" value="dd/MM/yyyy" />
</bean>
```

```
<bean name="aoi" class="org.geomajas.layer.hibernate.HibernateLayer">
    <property name="layerInfo" ref="aoiInfo" /> <!-- see previous section -->
    <property name="sessionFactory" ref="simpleSessionFactory" />

    <!-- Needed when the Hibernate pojo classes contain dates. -->
    <property name="dateFormat" ref="simpleDateFormat" />
</bean>
```

Note

This configuration only works for 1.9.0 and higher. Earlier versions also had to define the featureModel property as in this example (shown without the date format configuration).

```
<bean name="aoi" class="org.geomajas.layer.hibernate.HibernateLayer">
    <property name="layerInfo" ref="aoiInfo" />
    <property name="featureModel">
        <bean class="org.geomajas.layer.hibernate.HibernateFeatureModel">
            <property name="sessionFactory" ref="simpleSessionFactory" />
        </bean>
    </property>
    <property name="sessionFactory" ref="simpleSessionFactory" />
</bean>
```

Note that in the example above, some extra configuration was added to support the use of java.util.Date objects as fields within Java pojo classes. In the case of the AreaOfInterest layer there was no such date, so technically this addition was not necessary.

The properties which may be defined on a HibernateLayer object are:

- *layerInfo*: the description of the features.
- *featureModel*: the feature model to use for this layer.
- *sessionfactory*: the session factory to use for this layer.
- *scrollableResultSet*: indicates whether a scrollable resultset needs to be used. This can be more efficient when queries return many records but needs to be supported by your database driver.
- *useLazyFeatureConversion*: indicates whether lazy feature conversion should be used. This is set to true by default. You should only set this to false if none of attributes in the feature are lazy loaded.

Note

You have now successfully created a Geomajas VectorLayer definition, using the Hibernate layer plug-in. All you have to do now, is use it in your map configuration.

Important

This tutorial only covered the most basic case of mapping a single table onto a Java class. More complex mappings (many-to-one and one-to-many) are also supported. See the section Chapter 3, *How-to* for more information.

Chapter 3. How-to

This section covers a few specific cases that might come in handy when really using the Geomajas Hibernate layer plug-in.

1. How to use a many-to-one relation

A many-to-one relation is the Hibernate term for what would in the database world be called a foreign key. Say you have a table ("areaofinterest") with a foreign key to some other table("aoitype"). In order to get this relation configured as an attribute within the Geomajas layer definition, the following must be done:

- Both tables must actually exist in the database.
- For both tables a Java O/R mapping class must be defined.
- The Geomajas layer definition must include the many-to-one relation in it's attribute definitions.

1.1. Spatial database tables

This time, 2 tables must be present in the database, in order for one to be able to point to the other. Let's say the second table, containing the type, is very simple and holds only an ID and a description:

Example 3.1. SQL for creating a PostGIS spatial table

```
CREATE TABLE aoitype (
    id integer NOT NULL,
    description character varying,
);
```

Now we have the "areaofinterest" table point to it with a foreign key ("type_id"):

Example 3.2. SQL for creating a PostGIS spatial table

```
CREATE TABLE areaofinterest (
    id integer NOT NULL,
    title character varying(100) NOT NULL,
    description character varying,
    type_id integer NOT NULL,
    geom geometry,
    CONSTRAINT enforce_dims_geom CHECK ((ndims(geom) = 2)),
    CONSTRAINT enforce_geotype_geom CHECK (((geometrytype(geom) = 'POLYGON'::te
    CONSTRAINT enforce_srid_geom CHECK ((srid(geom) = 900913))
);

ALTER TABLE ONLY areaofinterest ADD CONSTRAINT fk_areaofinterest_aoitype FOREIG
```

1.2. Java O/R mapping

For both database table, we will now create Java mapping classes. First the "aoitype":

Example 3.3. AoiType class

```
@Entity
@Table(name = "aoitype")
public class AoiType{
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "description")
    private String description;

    // Constructors, getters, and setters
    ....
```

We now update the AreaOfInterest class to include the ManyToOne relation:

Example 3.4. AreaOfInterest class

```
@Entity
@Table(name = "areaofinterest")
public class AreaOfInterest {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, name = "title")
    private String title;

    @Column(name = "description")
    private String description;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "type_id", nullable = false)
    private AoiType type;

    @Type(type = "org.hibernatespatial.GeometryUserType")
    @Column(nullable = false, name = "geom")
    private Geometry geometry;

    // Constructors, getters, and setters
    ....
```

1.3. Vector layer configuration

Lastly, you add the new many-to-one relation to the list of attributes:

Example 3.5. VectorLayerInfo for the AreaOfInterest class

```
...
    <property name="attributes">
        <list>
            <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
                <property name="label" value="Title" />
                <property name="name" value="title" />
                <property name="editable" value="true" />
                <property name="identifying" value="true" />
                <property name="type" value="STRING" />
            </bean>
            <bean class="org.geomajas.configuration.PrimitiveAttributeInfo">
                <property name="label" value="Description" />
```

```
                    <property name="name" value="description" />
                    <property name="editable" value="true" />
                    <property name="identifying" value="false" />
                    <property name="type" value="STRING" />
                </bean>

                <bean class="org.geomajas.configuration.AssociationAttributeInfo">
                    <property name="label" value="Type" />
                    <property name="name" value="type" />
                    <property name="editable" value="true" />
                    <property name="identifying" value="false" />
                    <property name="type" value="MANY_TO_ONE" />
                    <property name="feature">
                        <bean class="org.geomajas.configuration.FeatureInfo">
                            <property name="dataSourceName" value="org.geomajas.serv
                            <property name="identifier">
                                <bean class="org.geomajas.configuration.PrimitiveAt
                                    <property name="label" value="Id" />
                                    <property name="name" value="id" />
                                    <property name="type" value="LONG" />
                                </bean>
                            </property>
                            <property name="attributes">
                                <list>
                                    <bean class="org.geomajas.configuration.Primiti
                                        <property name="label" value="Description"
                                        <property name="name" value="description" />
                                        <property name="editable" value="false" />
                                        <property name="identifying" value="true" />
                                        <property name="type" value="STRING" />
                                    </bean>
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>

            </list>
        </property>
...
```

2. How to use a one-to-many relation

TODO

3. How to create my own DAOs

If you have followed the configuration guidelines and made use of the Spring configuration options, this will be a piece of cake: we inject the SessionFactory at run-time into your DAO implementation, and this SessionFactory will take care of all session and transaction handling.

Say, for example, we have the following DAO interface:

Example 3.6. DAO interface

```
public interface ZoneDao {
```

```
    List<AreaOfInterest> getByTitle(String title);
}
```

In this case, a possible implementation could look like this:

Example 3.7. DAO implementation

```
@Component
@Transactional(rollbackFor = Throwable.class, propagation = Propagation.REQUIRE
public class ZoneDaoImpl implements ZoneDao {

    @Autowired
    private SessionFactory sessionFactory;

    public List<AreaOfInterest> getByTitle(String title) {
        Session session = sessionFactory.getCurrentSession();
        Query query = session.createQuery("FROM AreaOfInterest where title LIKE
        return (List<AreaOfInterest>) query.list();
    }
}
```

And that's it! No more worrying about sessions or transactions, or how the hell everything should get initialized...

4. How to use scrollable resultsets

If you have a very large table it might be desirable to retrieve features not as a list but as a scrollable resultset so only the features you actually use are also retrieved from the underlying database (for instance when paging).

To retrieve features as a scrollable resultset you add the property scrollableResultSet to your hibernate layer definition:

Example 3.8. Hibernate layer definition

```
<bean name="midori" class="org.geomajas.layer.hibernate.HibernateLayer">
    <property name="layerInfo" ref="midoriInfo" />
    <property name="scrollableResultSet" value="true" />
    <property name="featureModel">
        <bean class="org.geomajas.layer.hibernate.HibernateFeatureModel">
            <property name="sessionFactory" ref="simpleSessionFactory" />
        </bean>
    </property>
    <property name="sessionFactory" ref="simpleSessionFactory" />
</bean>
```

Please note that your databasedriver needs to support Scrollable resultsets.

5. How to configure a connection pool

There are many connection pool libraries which can be used. Some of the best known include DBCP and C3P0. The former may cause deadlocks while the latter is not known for its speed. An alternative connection pool library is BoneCP [http://jolbox.com/]. This can be included using the following dependency:

```
<dependency>
    <groupId>com.jolbox</groupId>
```

```
    <artifactId>bonecp</artifactId>
</dependency>
```

A sample Hibernate definition looks like this:

```
<!-- BoneCP configuration -->
<bean id="postgisDataSource" class="com.jolbox.bonecp.BoneCPDataSource" destroy
   <property name="driverClass" value="org.postgresql.Driver" />
   <property name="jdbcUrl" value="jdbc:postgresql://localhost:5432/databaseNam
   <property name="username" value="dbUser"/>
   <property name="password" value="dbPw"/>
   <property name="idleConnectionTestPeriod" value="60"/>
   <property name="idleMaxAge" value="240"/>
   <property name="maxConnectionsPerPartition" value="30"/>
   <property name="minConnectionsPerPartition" value="10"/>
   <property name="partitionCount" value="3"/>
   <property name="acquireIncrement" value="5"/>
   <property name="statementsCacheSize" value="100"/>
   <property name="releaseHelperThreads" value="3"/>
</bean>

<!-- Hibernate SessionFactory -->
<bean id="postgisSessionFactory" class="org.springframework.orm.hibernate3.Loca
    <property name="dataSource" ref="postgisDataSource" />
    <property name="configLocation" value="classpath:/hibernate.cfg.xml" />
    <property name="configurationClass" value="org.hibernate.cfg.AnnotationConf
</bean>

<!-- enable the configuration of transactional behavior based on annotations --
<tx:annotation-driven transaction-manager="transactionManager" />

<bean id="transactionManager" class="org.springframework.orm.hibernate3.Hiberna
    <property name="sessionFactory" ref="postgisSessionFactory" />
</bean>

<!-- Needed when the Hibernate pojo classes contain dates. -->
<bean name="simpleDateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg type="java.lang.String" value="dd/MM/yyyy" />
</bean
```

6. How to use a second level cache

In your hibernate.cfg.xml file, add the following excerpt:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.infinispan.InfinispanRegionFactory
</property>
```

To make sure this works, you also need the Infinispan [http://www.jboss.org/infinispan/] dependencies (these are already available when using the caching plug-in. Additionally, you also need the Hibernate-Infinispan bridge:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-infinispan</artifactId>
</dependency>
```

7. Error "Operation on two geometries with different SRIDs"

This is an exception which can occur of your geometries are stored in the database using a different SRID as the one configured in the layer.

You have to make sure that the SRID mathes the declaration.

When converting a shapefile using shp2pqsql, you have to specify the SRID to be set (as this defaults to -1). Use a command like:

```
shp2pgsql -s 4326 shape-base-name
```