HIBERNATE - Relational Persistence for Idiomatic Java

1

Hibernate Reference Documentation

3.6.7.Final

King Gavin [FAMILY Given], Bauer Christian [FAMILY Given], Andersen Max [FAMILY Given], Bernard Emmanuel [FAMILY Given], Ebersole Steve [FAMILY Given], # Ferentschik Hardy [FAMILY Given]

and thanks to Cobb James [FAMILY Given] (Graphic Design) # Weaver Cheyenne [FAMILY Given] (Graphic Design)

###

Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming. Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases. Hibernate is an Object/Relational Mapping solution for Java environments. The term Object/Relational Mapping refers to the technique of mapping data from an object model representation to a relational data model representation (and visa versa). See *http://en.wikipedia.org/wiki/Object-relational_mapping* for a good high-level discussion.

##

While having a strong background in SQL is not required to use Hibernate, having a basic understanding of the concepts can greatly help you understand Hibernate more fully and quickly. Probably the single best background is an understanding of data modeling principles. You might want to consider these resources as a good starting point:

- *http://www.agiledata.org/essays/dataModeling101.html*
- *http://en.wikipedia.org/wiki/Data_modeling*

Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC. However, unlike many other persistence solutions, Hibernate does not hide the power of SQL from you and guarantees that your investment in relational technology and knowledge is as valid as always.

Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

Hibernate ########/############## #### Java ######## ##############

1. Read *1#Tutorial* for a tutorial with step-by-step instructions. The source code for the tutorial is included in the distribution in the `doc/reference/tutorial/` directory.
2. Read *2########* to understand the environments where Hibernate can be used.
3. Hibernate ############## `eg/` ############### ############################ ### #### JDBC ##### `lib/` ## `etc/`

`hibernate.properties` ####### ################################ `ant eg` (Ant ###)## ##### Windows ###### `build eg` #######

4. Use this reference documentation as your primary source of information. Consider reading [*JPwH*] if you need more help with application design, or if you prefer a step-by-step tutorial. Also visit *http://caveatemptor.hibernate.org* and download the example application from [*JPwH*].

5. ########### (FAQ) # Hibernate ###############

6. Links to third party demos, examples, and tutorials are maintained on the Hibernate website.

7. Hibernate ######## Community Area ########################## (Tomcat# JBoss AS# Struts# EJB ##)######################

There are a number of ways to become involved in the Hibernate community, including

- Trying stuff out and reporting bugs. See *http://hibernate.org/issuetracker.html* details.
- Trying your hand at fixing some bugs or implementing enhancements. Again, see *http://hibernate.org/issuetracker.html* details.
- *http://hibernate.org/community.html* list a few ways to engage in the community.
  - There are forums for users to ask questions and receive help from the community.
  - There are also *IRC* [http://en.wikipedia.org/wiki/Internet_Relay_Chat] channels for both user and developer discussions.
- Helping improve or translate this documentation. Contact us on the developer mailing list if you have interest.
- Evangelizing Hibernate within your organization.

Tutorial

Intended for new users, this chapter provides an step-by-step introduction to Hibernate, starting with a simple application using an in-memory database. The tutorial is based on an earlier tutorial developed by Michael Gloegl. All code is contained in the `tutorials/web` directory of the project source.

####

This tutorial expects the user have knowledge of both Java and SQL. If you have a limited knowledge of JAVA or SQL, it is advised that you start with a good introduction to that technology prior to attempting to learn Hibernate.

##

The distribution contains another example application under the `tutorial/eg` project source directory.

1.1. ###1 - #### Hibernate

##
######

##

Although you can use whatever database you feel comfortable using, we will use *HSQLDB* [http://hsqldb.org/] (an in-memory, Java database) to avoid describing installation/setup of any particular database servers.

1.1.1. Setup

The first thing we need to do is to set up the development environment. We will be using the "standard layout" advocated by alot of build tools such as *Maven* [http://maven.org]. Maven, in particular, has a good resource describing this *layout* [http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html]. As this tutorial is to be a web application, we will be creating and making use of `src/main/java`, `src/main/resources` and `src/main/webapp` directories.

We will be using Maven in this tutorial, taking advantage of its transitive dependency management capabilities as well as the ability of many IDEs to automatically set up a project for us based on the maven descriptor.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.hibernate.tutorials</groupId>
    <artifactId>hibernate-tutorial</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>First Hibernate Tutorial</name>

    <build>
         <!-- we dont want the version to be part of the generated war file name -->
         <finalName>${artifactId}</finalName>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
        </dependency>

        <!-- Because this is a web app, we also have a dependency on the servlet api. -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
        </dependency>

        <!-- Hibernate uses slf4j for logging, for our purposes here use the simple backend -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
        </dependency>

        <!-- Hibernate gives you a choice of bytecode providers between cglib and javassist -->
        <dependency>
            <groupId>javassist</groupId>
            <artifactId>javassist</artifactId>
        </dependency>
    </dependencies>

</project>
```

####

It is not a requirement to use Maven. If you wish to use something else to build this tutorial (such as Ant), the layout will remain the same. The only change is that you will need to manually account for all the needed dependencies. If you use something like *Ivy* [http://ant.apache.org/ivy/] providing transitive dependency management you would still use the dependencies mentioned below. Otherwise, you'd need to grab *all* dependencies, both explicit and transitive, and add them to the project's classpath. If working from the Hibernate distribution bundle, this

would mean `hibernate3.jar`, all artifacts in the `lib/required` directory and all files from either the `lib/bytecode/cglib` or `lib/bytecode/javassist` directory; additionally you will need both the servlet-api jar and one of the slf4j logging backends.

Save this file as `pom.xml` in the project root directory.

1.1.2.

##############################

```
package org.hibernate.tutorial.domain;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

This class uses standard JavaBean naming conventions for property getter and setter methods, as well as private visibility for the fields. Although this is the recommended design, it is not required. Hibernate can also access fields directly, the benefit of accessor methods is robustness for refactoring.

id ##################################### Hibernate ############################## # ########################## ### # ## web # ######## ### ID ## ########################## private ######## Hibernate ########################## ##### Hibernate ##public, private, protected###################### public, private, protected ###

################################ Hibernate # Java ################################ ############ private ## package # ########

Save this file to the `src/main/java/org/hibernate/tutorial/domain` directory.

1.1.3.

Hibernate ## Hibernate ############ ## Hibernate ##### ##

############################

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.domain">
[...]
</hibernate-mapping
>
```

Hibernate DTD ########################## DTD ####### IDE ## XML ################ ################ DTD ## ################## Hibernate ## web ## DTD ################################### ################ DTD ##### Hibernate ########### `src/` ##########`hibernate3.jar` ##########

####

################## DTD ################################

2## `hibernate-mapping` ##### `class` ## ############################## SQL ################################

```
<hibernate-mapping package="org.hibernate.tutorial.domain">
```

```
    <class name="Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

######### Event ########### EVENTS ################################ Hibernate ## Hibernate ########## #####

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

The `id` element is the declaration of the identifier property. The `name="id"` mapping attribute declares the name of the JavaBean property and tells Hibernate to use the `getId()` and `setId()` methods to access the property. The column attribute tells Hibernate which column of the `EVENTS` table holds the primary key value.

The nested `generator` element specifies the identifier generation strategy (aka how are identifier values generated?). In this case we choose `native`, which offers a level of portability depending on the configured database dialect. Hibernate supports database generated, globally unique, as well as application assigned, identifiers. Identifier value generation is also one of Hibernate's many extension points and you can plugin in your own strategy.

> **####**
>
> `native` is no longer consider the best strategy in terms of portability. for further discussion, see *#Identifier generation#*

##

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
```

```
        <property name="title"/>
    </class>

</hibernate-mapping>
```

`id` ########## `property` ### `name` ########################## Hibernate ############ Hibernate # `getDate()/setDate()` # `getTitle()/setTitle()` ######

> ##
>
> ## `date` ############# `column` ###### `title` ################ `column` #### #### Hibernate ############################## `title` ############# `date` ##

######## `title` ###### `type` ############################# `type` ############### Java ############## SQL #################### *Hibernate*###### ###### Java ## SQL ######## # SQL ## Java ############################# Hibernate # `type` ###################### ############################## #Java## ####################### `date` ############### Hibernate #### `java.util.Date` ####### # SQL # `date` , `timestamp` , `time` ############################## `timestamp` ########### ###########################

> ####
>
> Hibernate makes this mapping type determination using reflection when the mapping files are processed. This can take time and resources, so if startup performance is important you should consider explicitly defining the type to use.

Save this mapping file as `src/main/resources/org/hibernate/tutorial/domain/Event.hbm.xml`.

1.1.4. Hibernate

At this point, you should have the persistent class and its mapping file in place. It is now time to configure Hibernate. First let's set up HSQLDB to run in "server mode"

> ##
>
> We do this do that the data remains between runs.

We will utilize the Maven exec plugin to launch the HSQLDB server by running: `mvn exec:java -Dexec.mainClass="org.hsqldb.Server" -Dexec.args="-database.0 file:target/data/tutorial"` You will see it start up and bind to a TCP/IP socket; this is where our application will

connect later. If you want to start with a fresh database during this tutorial, shutdown HSQLDB, delete all files in the `target/data` directory, and start HSQLDB again.

Hibernate will be connecting to the database on behalf of your application, so it needs to know how to obtain connections. For this tutorial we will be using a standalone connection pool (as opposed to a `javax.sql.DataSource`). Hibernate comes with support for two third-party open source JDBC connection pools: *c3p0* [https://sourceforge.net/projects/c3p0] and *proxool* [http://proxool.sourceforge.net/]. However, we will be using the Hibernate built-in connection pool for this tutorial.

##

The built-in Hibernate connection pool is in no way intended for production use. It lacks several features found on any decent connection pool.

Hibernate ######### `hibernate.properties` ################### `hibernate.cfg.xml` ### ## XML #########

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class"
>org.hsqldb.jdbcDriver</property>
        <property name="connection.url"
>jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username"
>sa</property>
        <property name="connection.password"
></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size"
>1</property>

        <!-- SQL dialect -->
        <property name="dialect"
>org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class"
>thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class"
>org.hibernate.cache.NoCacheProvider</property>
```

```
        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql"
>true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto"
>update</property>

        <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration
>
```

##

XML ####### DTD

############################ Hibernate # `SessionFactory` ########################## ## ###################################### `<session-factory>` ##############

###4## `property` ### JDBC ##################### dialect ###### `property` #### Hibernate ######## SQL #########

####

In most cases, Hibernate is able to properly determine which dialect to use. See *#Dialect resolution#* for more information.

############## Hibernate ## `hbm2ddl.auto` ## ###################### on ################################config ########### off ### ## `SchemaExport` ### Ant ## #####

Save this file as `hibernate.cfg.xml` into the `src/main/resources` directory.

1.1.5. Maven

We will now build the tutorial with Maven. You will need to have Maven installed; it is available from the *Maven download page* [http://maven.apache.org/download.html]. Maven will read the `/pom.xml` file we created earlier and know how to perform some basic project tasks. First, lets run the `compile` goal to make sure we can compile everything so far:

```
[hibernateTutorial]$ mvn compile
```

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building First Hibernate Tutorial
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /home/steve/projects/sandbox/hibernateTutorial/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jun 09 12:25:25 CDT 2009
[INFO] Final Memory: 5M/547M
[INFO] ------------------------------------------------------------------------
```

1.1.6.

Event ## ######## Hibernate ## SessionFactory ######## ## org.hibernate.SessionFactory #### org.hibernate.Session ############### org.hibernate.Session ###############(Unit of Work)############ org.hibernate.SessionFactory ##################################

################## org.hibernate.SessionFactory ####### HibernateUtil ########### ##############

```
package org.hibernate.tutorial.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
```

```
}
```

Save this code as `src/main/java/org/hibernate/tutorial/util/HibernateUtil.java`

This class not only produces the global `org.hibernate.SessionFactory` reference in its static initializer; it also hides the fact that it uses a static singleton. We might just as well have looked up the `org.hibernate.SessionFactory` reference from JNDI in an application server or any other location for that matter.

If you give the `org.hibernate.SessionFactory` a name in your configuration, Hibernate will try to bind it to JNDI under that name after it has been built. Another, better option is to use a JMX deployment and let the JMX-capable container instantiate and bind a `HibernateService` to JNDI. Such advanced options are discussed later.

You now need to configure a logging system. Hibernate uses commons logging and provides two choices: Log4j and JDK 1.4 logging. Most developers prefer Log4j: copy `log4j.properties` from the Hibernate distribution in the `etc/` directory to your `src` directory, next to `hibernate.cfg.xml`. If you prefer to have more verbose output than that provided in the example configuration, you can change the settings. By default, only the Hibernate startup message is shown on stdout.

################## Hibernate ####################

1.1.7.

We are now ready to start doing some real work with Hibernate. Let's start by writing an `EventManager` class with a `main()` method:

```
package org.hibernate.tutorial;

import org.hibernate.Session;

import java.util.*;

import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
```

```
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

In `createAndStoreEvent()` we created a new `Event` object and handed it over to Hibernate. At that point, Hibernate takes care of the SQL and executes an `INSERT` on the database.

A org.hibernate.Session is designed to represent a single unit of work (a single atomic piece of work to be performed). For now we will keep things simple and assume a one-to-one granularity between a Hibernate org.hibernate.Session and a database transaction. To shield our code from the actual underlying transaction system we use the Hibernate `org.hibernate.Transaction` API. In this particular case we are using JDBC-based transactional semantics, but it could also run with JTA.

What does `sessionFactory.getCurrentSession()` do? First, you can call it as many times and anywhere you like once you get hold of your `org.hibernate.SessionFactory`. The `getCurrentSession()` method always returns the "current" unit of work. Remember that we switched the configuration option for this mechanism to "thread" in our `src/main/resources/hibernate.cfg.xml`? Due to that setting, the context of a current unit of work is bound to the current Java thread that executes the application.

####

Hibernate offers three methods of current session tracking. The "thread" based method is not intended for production use; it is merely useful for prototyping and tutorials such as this one. Current session tracking is discussed in more detail later on.

A org.hibernate.Session begins when the first call to `getCurrentSession()` is made for the current thread. It is then bound by Hibernate to the current thread. When the transaction ends, either through commit or rollback, Hibernate automatically unbinds the org.hibernate.Session from the thread and closes it for you. If you call `getCurrentSession()` again, you get a new org.hibernate.Session and can start a new unit of work.

(Unit of Work) ######## Hibernate # org.hibernate.Session #1###################### ##1## org.hibernate.Session ################# ########################### Hibernate # org.hibernate.Session ########## ### ######## ############ Hibernate org.hibernate.Session ## ####### (###) ####################### ######### *Session* ############# (###) #######

See *13#Transactions and Concurrency* for more information about transaction handling and demarcation. The previous example also skipped any error handling and rollback.

To run this, we will make use of the Maven exec plugin to call our class with the necessary classpath setup: `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="store"`

##

You may need to perform `mvn compile` first.

######### Hibernate ##

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

This is the `INSERT` executed by Hibernate.

To list stored events an option is added to the main method:

```
        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }
        else if (args[0].equals("list")) {
            List events = mgr.listEvents();
            for (int i = 0; i < events.size(); i++) {
                Event theEvent = (Event) events.get(i);
                System.out.println(
                        "Event: " + theEvent.getTitle() + " Time: " + theEvent.getDate()
                );
            }
        }
```

`listEvents()`####

```
    private List listEvents() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        List result = session.createQuery("from Event").list();
        session.getTransaction().commit();
        return result;
    }
```

Here, we are using a Hibernate Query Language (HQL) query to load all existing `Event` objects from the database. Hibernate will generate the appropriate SQL, send it to the database and populate `Event` objects with the data. You can create more complex queries with HQL. See *16 #HQL: Hibernate #####* for more information.

Now we can call our new functionality, again using the Maven exec plugin: `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="list"`

1.2. ###2 -

##
######################

1.2.1. Person

Person

```
package org.hibernate.tutorial.domain;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

Save this to a file named `src/main/java/org/hibernate/tutorial/domain/Person.java`

Next, create the new mapping file as `src/main/resources/org/hibernate/tutorial/domain/Person.hbm.xml`

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

Hibernate

```
<mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>
```

```
<mapping resource="org/hibernate/tutorial/domain/Person.hbm.xml"/>
```

#######2###
######################

1.2.2. ### Set

By adding a collection of events to the `Person` class, you can easily navigate to the events for a particular person, without executing an explicit query - by calling `Person#getEvents`. Multi-valued associations are represented in Hibernate by one of the Java Collection Framework contracts; here we choose a `java.util.Set` because the collection will not contain duplicate elements and the ordering is not relevant to our examples:

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

Before mapping this association, let's consider the other side. We could just keep this unidirectional or create another collection on the `Event`, if we wanted to be able to navigate it from both directions. This is not necessary, from a functional perspective. You can always execute an explicit query to retrieve the participants for a particular event. This is a design choice left to you, but what is clear from this discussion is the multiplicity of the association: "many" valued on both sides is called a *many-to-many* association. Hence, we use Hibernate's many-to-many mapping:

```
<class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="Event"/>
    </set>

</class>
```

Hibernate ## set ### ######### *n:m* # ## set ### table ######################## key ############### many-to-many # column ######### Hibernate ################## ######################################

###################################

```
    _____________       __________________
   |             |     |                  |       _____________
   |   EVENTS    |     |   PERSON_EVENT   |      |             |
   |_____________|     |__________________|      |    PERSON   |
   |             |     |                  |      |_____________|
   | *EVENT_ID   | <--> | *EVENT_ID       |      |             |
   |  EVENT_DATE |     | *PERSON_ID       | <--> | *PERSON_ID  |
   |  TITLE      |     |__________________|      |  AGE        |
   |_____________|                               |  FIRSTNAME  |
                                                 |  LASTNAME   |
                                                 |_____________|
```

1.2.3.

EventManager ##########################

```
private void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);
    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

After loading a `Person` and an `Event`, simply modify the collection using the normal collection methods. There is no explicit call to `update()` or `save()`; Hibernate automatically detects that the collection has been modified and needs to be updated. This is called *automatic dirty checking*. You can also try it by modifying the name or the date property of any of your objects. As long as they are in *persistent* state, that is, bound to a particular Hibernate `org.hibernate.Session`, Hibernate monitors any changes and executes SQL in a write-behind fashion. The process of synchronizing the memory state with the database, usually only at the end of a unit of work, is called *flushing*. In our code, the unit of work ends with a commit, or rollback, of the database transaction.

####### (Unit of Work) ## ## ###*detached*# ####### `org.hibernate.Session` ################################### ###########

```
    private void addPersonToEvent(Long personId, Long eventId) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Person aPerson = (Person) session
               .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
                .setParameter("pid", personId)
                .uniqueResult(); // Eager fetch the collection so we can use it detached
        Event anEvent = (Event) session.load(Event.class, eventId);

        session.getTransaction().commit();

        // End of first unit of work

        aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

        // Begin second unit of work

        Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
        session2.beginTransaction();
        session2.update(aPerson); // Reattachment of aPerson

        session2.getTransaction().commit();
    }
```

update ## (Unit of Work) ######################## ##

EventManager ## save() ############

```
        else if (args[0].equals("addpersontoevent")) {
            Long eventId = mgr.createAndStoreEvent("My Event", new Date());
            Long personId = mgr.createAndStorePerson("Foo", "Bar");
            mgr.addPersonToEvent(personId, eventId);
            System.out.println("Added person " + personId + " to event " + eventId);
        }
```

###########2#########2### #################### int # java.lang.String ################### ## ############## ############ ## ################### ID ############################## ########## ##########2####### firstname ####################### JDK ################### #### Hibernate ################ JDK ############## ### Address # MonetaryAmount ###### #################

Java ############

1.2.4.

Let's add a collection of email addresses to the `Person` entity. This will be represented as a `java.util.Set` of `java.lang.String` instances:

```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

`Set`

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

############### `element` ####### Hibernate ########################### `string` # ############################ (string) # Hibernate ################################# ###`set` ### `table` ########################## `key` ############################# `element` ### `column` ### `string` ########################

################

```
 _____________        __________________
|             |      |                  |     _____________
|   EVENTS    |      |   PERSON_EVENT   |    |             |       ___________________
|_____________|      |__________________|    |    PERSON   |      |                   |
|             |      |                  |    |_____________|      | PERSON_EMAIL_ADDR |
| *EVENT_ID   | <--> | *EVENT_ID        |    |             |      |___________________|
|  EVENT_DATE |      | *PERSON_ID       | <--> | *PERSON_ID  | <--> |  *PERSON_ID        |
|  TITLE      |      |__________________|    |  AGE        |      |  *EMAIL_ADDR       |
|_____________|                              |  FIRSTNAME  |      |___________________|
                                             |  LASTNAME   |
                                             |_____________|
```

E ##################### Java # set

Java ########

```
    private void addEmailToPerson(Long personId, String emailAddress) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Person aPerson = (Person) session.load(Person.class, personId);
        // adding to the emailAddress collection might trigger a lazy load of the collection
        aPerson.getEmailAddresses().add(emailAddress);

        session.getTransaction().commit();
    }
```

This time we did not use a *fetch* query to initialize the collection. Monitor the SQL log and try to optimize this with an eager fetch.

1.2.5.

################## Java ##
####

##

###
############################

Event

```
    private Set participants = new HashSet();

    public Set getParticipants() {
        return participants;
    }

    public void setParticipants(Set participants) {
        this.participants = participants;
    }
```

Event.hbm.xml

```
        <set name="participants" table="PERSON_EVENT" inverse="true">
            <key column="EVENT_ID"/>
            <many-to-many column="PERSON_ID" class="Person"/>
        </set
>
```

########## (XML####) ###### `set` ############ `key` # `many-to-many` ####
`Event` ############ `set`
`inverse="true"`

#########2################################## Hibernate ############### `Person` ###
##############2##

1.2.6.

Hibernate #### Java
`Person` # `Event` ################ `Person` ######################## `Event` ##########
`Event` ######## `Person`
##

`Person`
####

```
    protected Set getEvents() {
        return events;
    }

    protected void setEvents(Set events) {
        this.events = events;
    }

    public void addToEvent(Event event) {
        this.getEvents().add(event);
        event.getParticipants().add(this);
    }

    public void removeFromEvent(Event event) {
        this.getEvents().remove(event);
        event.getParticipants().remove(this);
    }
```

########## protected ##
########## ###### ##
##############

`inverse` ######################## Java ##
Hibernate ############## SQL # `INSERT` # `UPDATE` ###################################
############################## `inverse` ######### Hibernate ################### #
############## Hibernate ################# SQL ###################################
`inverse`
##

1.3. ###3 - EventManager Web

Hibernate # Web ################################ `Session` # `Transaction` ########## ######################## `EventManagerServlet` ################################## ########## HTML ########################

1.3.1. #### Servlet

Servlet # HTTP # `GET` ################ `doGet()` #######

```
package org.hibernate.tutorial.web;

// Imports

public class EventManagerServlet extends HttpServlet {

    protected void doGet(
            HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        SimpleDateFormat dateFormatter = new SimpleDateFormat( "dd.MM.yyyy" );

        try {
            // Begin unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().beginTransaction();

            // Process request and render page...

            // End unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().commit();
        }
        catch (Exception ex) {
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().rollback();
            if ( ServletException.class.isInstance( ex ) ) {
                throw ( ServletException ) ex;
            }
            else {
                throw new ServletException( ex );
            }
        }
    }

}
```

Save this servlet as `src/main/java/org/hibernate/tutorial/web/EventManagerServlet.java`

session-per-request ########## Servlet ############# `SessionFactory` # `getCurrentSession()` ########## Hibernate #### `Session` ############################ ##

###################### Hibernate `Session` ### ######## ################## Hibernate `Session` ################## Java ################ `getCurrentSession()` ##########

############################## HTML ########################

############ HTML ################ (Unit of Work) ################################ ################################## `session-per-request` ########################## ### *Open Session in View* ################ Hibernate # Web #### Wiki ##################### JSP # HTML ###################### ################

1.3.2.

#########################

```
        // Write HTML header
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Event Manager</title></head><body>");

        // Handle actions
        if ( "store".equals(request.getParameter("action")) ) {

            String eventTitle = request.getParameter("eventTitle");
            String eventDate = request.getParameter("eventDate");

            if ( "".equals(eventTitle) || "".equals(eventDate) ) {
                out.println("<b><i>Please enter event title and date.</i></b>");
            }
            else {
                createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
                out.println("<b><i>Added event.</i></b>");
            }
        }

        // Print page
       printEventForm(out);
       listEvents(out, dateFormatter);

       // Write HTML footer
       out.println("</body></html>");
       out.flush();
       out.close();
```

Java # HTML ## ################ Hibernate ## HTML ######################## ######### HTML ## HTML #####

```
    private void printEventForm(PrintWriter out) {
        out.println("<h2>Add new event:</h2>");
        out.println("<form>");
        out.println("Title: <input name='eventTitle' length='50'/><br/>");
        out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
```

```
        out.println("<input type='submit' name='action' value='store'/>");
        out.println("</form>");
    }
```

`listEvents()` ################## Hibernate # `Session` ################

```
    private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

        List result = HibernateUtil.getSessionFactory()
                .getCurrentSession().createCriteria(Event.class).list();
        if (result.size() > 0) {
            out.println("<h2>Events in database:</h2>");
            out.println("<table border='1'>");
            out.println("<tr>");
            out.println("<th>Event title</th>");
            out.println("<th>Event date</th>");
            out.println("</tr>");
            Iterator it = result.iterator();
            while (it.hasNext()) {
                Event event = (Event) it.next();
                out.println("<tr>");
                out.println("<td>" + event.getTitle() + "</td>");
                out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
                out.println("</tr>");
            }
            out.println("</table>");
        }
    }
```

`store` ###### `createAndStoreEvent()` ############################## `Session` ###

```
    protected void createAndStoreEvent(String title, Date theDate) {
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        HibernateUtil.getSessionFactory()
                .getCurrentSession().save(theEvent);
    }
```

################################1## `Session` # `Transaction` ######################## ############## Hibernate ## ############### `SessionFactory` ## ################################DAO############### Hibernate # Wiki ##########

1.3.3.

To deploy this application for testing we must create a Web ARchive (WAR). First we must define the WAR descriptor as `src/main/webapp/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>org.hibernate.tutorial.web.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

########## `mvn package` ###### `hibernate-tutorial.war` ##### Tomcat # `webapp` ##########

##

If you do not have Tomcat installed, download it from *http://tomcat.apache.org/* and follow the installation instructions. Our application requires no changes to the standard Tomcat configuration.

######## Tomcat ####### `http://localhost:8080/hibernate-tutorial/eventmanager` ## ## Tomcat #### Hibernate ########## #### # `HibernateUtil` ##

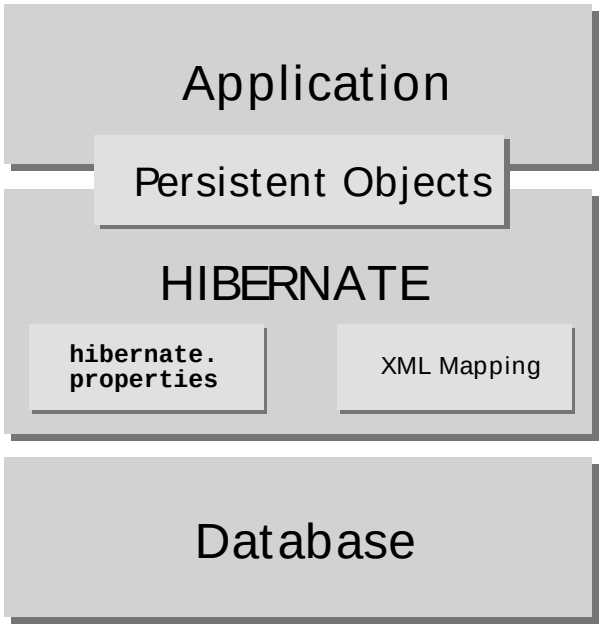
1.4.

########################## Hibernate ############## Web ########################

#######

2.1.

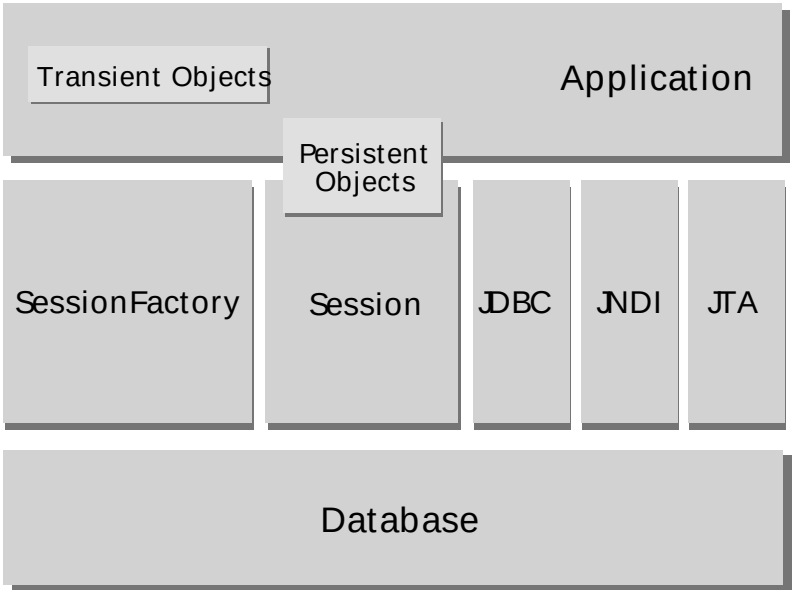
Hibernate ##########################



Unfortunately we cannot provide a detailed view of all possible runtime architectures. Hibernate is sufficiently flexible to be used in a number of ways in many, many architectures. We will, however, illustrate 2 specifically since they are extremes.

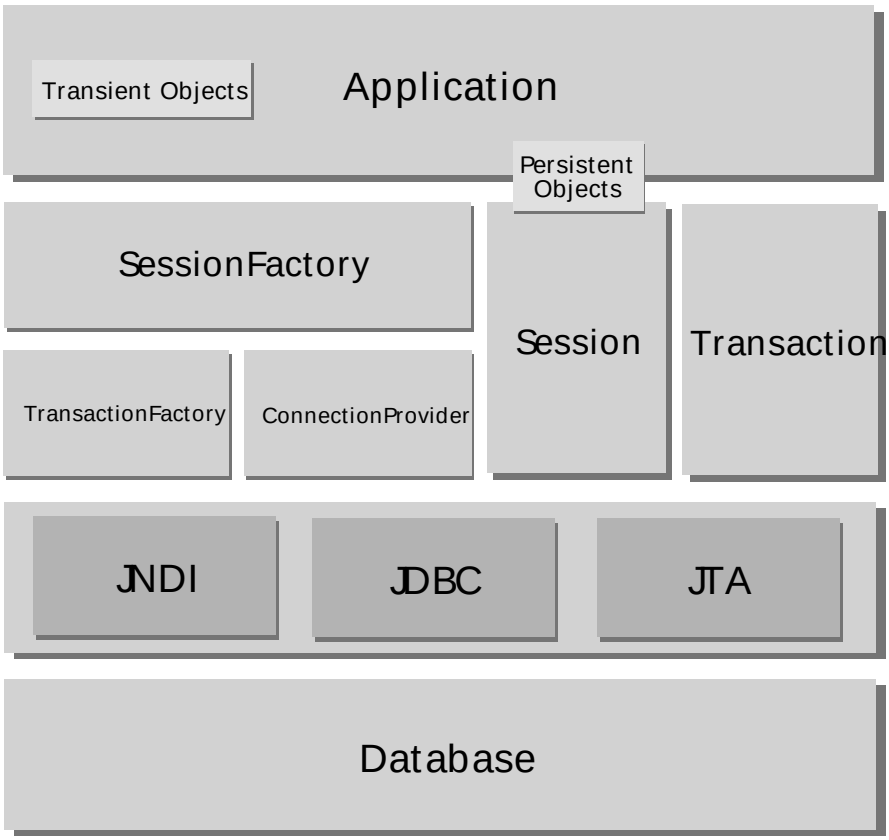
2.1.1. Minimal architecture

The "minimal" architecture has the application manage its own JDBC connections and provide those connections to Hibernate; additionally the application manages transactions for itself. This approach uses a minimal subset of Hibernate APIs.



2.1.2. Comprehensive architecture

The "comprehensive" architecture abstracts the application away from the underlying JDBC/JTA APIs and allows Hibernate to manage the details.



2.1.3. Basic APIs

Here are quick discussions about some of the API objects depicted in the preceding diagrams (you will see them again in more detail in later chapters).

SessionFactory (`org.hibernate.SessionFactory`)
: A thread-safe, immutable cache of compiled mappings for a single database. A factory for `org.hibernate.Session` instances. A client of `org.hibernate.connection.ConnectionProvider`. Optionally maintains a `second level cache` of data that is reusable between transactions at a process or cluster level.

Session (`org.hibernate.Session`)
: A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC `java.sql.Connection`. Factory for `org.hibernate.Transaction`. Maintains a `first level cache` of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

Persistent objects # Collections
: Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one `org.hibernate.Session`. Once the `org.hibernate.Session` is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation). *11##########* discusses transient, persistent and detached object states.

Transient # detached # objects # Collections
: Instances of persistent classes that are not currently associated with a `org.hibernate.Session`. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed `org.hibernate.Session`. *11########* discusses transient, persistent and detached object states.

Transaction (`org.hibernate.Transaction`)
: (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A `org.hibernate.Session` might span several `org.hibernate.Transactions` in some cases. However, transaction demarcation, either using the underlying API or `org.hibernate.Transaction`, is never optional.

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)
: (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying `javax.sql.DataSource` or `java.sql.DriverManager`. It is not exposed to application, but it can be extended and/or implemented by the developer.

TransactionFactory (`org.hibernate.TransactionFactory`)
: (Optional) A factory for `org.hibernate.Transaction` instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

Extension Interfaces

Hibernate ## API ######### #######

2.2. JMX

JMX # Java ########## J2EE ##### JMX ########### Hibernate ######################## `org.hibernate.jmx.HibernateService` ### MBean ###########

Another feature available as a JMX service is runtime Hibernate statistics. See *#Hibernate ###* for more information.

2.3.

Hibernate ## ## ##3.0#### Hibernate ###### `ThreadLocal` #### ######################## `HibernateUtil` ################## proxy/interception ######## ################ #Spring # Pico ###################################

Starting with version 3.0.1, Hibernate added the `SessionFactory.getCurrentSession()` method. Initially, this assumed usage of `JTA` transactions, where the `JTA` transaction defined both the scope and context of a current session. Given the maturity of the numerous stand-alone `JTA TransactionManager` implementations, most, if not all, applications should be using `JTA` transaction management, whether or not they are deployed into a `J2EE` container. Based on that, the `JTA`-based contextual sessions are all you need to use.

######### 3.1 #### `SessionFactory.getCurrentSession()` ######## ## (`org.hibernate.context.CurrentSessionContext`) ########### (`hibernate.current_session_context_class`) #########

See the Javadocs for the `org.hibernate.context.CurrentSessionContext` interface for a detailed discussion of its contract. It defines a single method, `currentSession()`, by which the implementation is responsible for tracking the current contextual session. Out-of-the-box, Hibernate comes with three implementations of this interface:

- `org.hibernate.context.JTASessionContext` - `JTA` ################################ ################ JTA ###################### Javadoc ##########
- `org.hibernate.context.ThreadLocalSessionContext` - ########################## ##### Javadoc ##########
- `org.hibernate.context.ManagedSessionContext` - ############################## ######## static ##### `Session` ##########/################################ `Session` ### ##################

The first two implementations provide a "one session - one database transaction" programming model. This is also known and used as *session-per-request*. The beginning and end of a Hibernate session is defined by the duration of a database transaction. If you use programmatic transaction demarcation in plain JSE without JTA, you are advised to use the Hibernate `Transaction` API to hide the underlying transaction system from your code. If you use JTA, you can utilize the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you do not need any transaction or session demarcation operations in your code. Refer to *13#Transactions and Concurrency* for more information and code examples.

`hibernate.current_session_context_class` ######### `org.hibernate.context.CurrentSessionContext` ## `org.hibernate.transaction.TransactionManagerLookup` ########### Hibernate # `org.hibernate.context.JTASessionContext` ################################3######## ##########################"jta"# "thread"# "managed"#####################

##

Hibernate ## Hibernate ####### ######## etc/ ##### hibernate.properties ################################## hibernate.properties ############################

3.1.

org.hibernate.cfg.Configuration ######### Java ### SQL ######################### Configuration ######## SessionFactory ################## XML ###################### ######

org.hibernate.cfg.Configuration ########### XML ########################### ### addResource()：

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

(################) ######################### Hibernate ############ #########

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate ##################################### /org/hibernate/auction/Item.hbm.xml # /org/hibernate/auction/Bid.hbm.xml ############################## ####

org.hibernate.cfg.Configuration ######################

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

Hibernate ##############1############################

1. java.util.Properties ####### Configuration.setProperties() ######
2. hibernate.properties ######################
3. System ###### java -Dproperty=value ############
4. <property> ### hibernate.cfg.xml ###########

If you want to get started quickly`hibernate.properties` is the easiest approach.

`org.hibernate.cfg.Configuration` ###################### `SessionFactory` ######### ###############

3.2. SessionFactory

When all mappings have been parsed by the `org.hibernate.cfg.Configuration`, the application must obtain a factory for `org.hibernate.Session` instances. This factory is intended to be shared by all application threads:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate does allow your application to instantiate more than one `org.hibernate.SessionFactory`. This is useful if you are using more than one database.

3.3. JDBC

####### `org.hibernate.SessionFactory` ##### SessionFactory # JDBC ################ ######################### `org.hibernate.Session` ############

```
Session session = sessions.openSession(); // open a new Session
```

############## JDBC ###################################

####### JDBC ############## Hibernate ################## Hibernate ############### `org.hibernate.cfg.Environment` ################## JDBC ####################

################## Hibernate ###################### `java.sql.DriverManager` #####:

#3.1 Hibernate JDBC

######	##
hibernate.connection.driver_class	*JDBC driver class*
hibernate.connection.url	*JDBC URL*
hibernate.connection.username	*########*
hibernate.connection.password	*###############*
hibernate.connection.pool_size	*maximum number of pooled connections*

Hibernate ## ############## ###### ## ######## `hibernate.connection.pool_size` ###################################### Hibernate ########################## C3P0 ######

C3P0 ######## JDBC ########### Hibernate # `lib` ################ `hibernate.c3p0.*` ############## Hibernate ## `C3P0ConnectionProvider` ######## Proxool ######### `hibernate.properties` ############ Hibernate # Web ######################

C3P0 ## `hibernate.properties` ###############

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

###################### Hibernate ## `javax.sql.Datasource` # JNDI ################################:

#3.2 Hibernate

######	##
hibernate.connection.datasource	####### *JNDI* #
hibernate.jndi.url	*JNDI* ###### *URL* (#####)
hibernate.jndi.class	*JNDI* #### *`InitialContextFactory`* (#####)
hibernate.connection.username	######### (#####)
hibernate.connection.password	################ (#####)

#################### JNDI ######### `hibernate.properties` ############

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

JNDI ############ JDBC ##

###################### "`hibernate.connnection`" ###################### charSet ###### ###### hibernate.connection.charSet ######

JDBC ################################## `org.hibernate.connection.ConnectionProvider` ############################ hibernate.connection.provider_class ##########

3.4.

################ Hibernate ##

##

Some of these properties are "system-level" only. System-level properties can be set only via `java -Dproperty=value` or `hibernate.properties`. They *cannot* be set by the other techniques described above.

#3.3 Hibernate

######	##
hibernate.dialect	Hibernate ##### `org.hibernate.dialect.Dialect` ######## ######################## SQL ####### `#full.classname.of.Dialect` In most cases Hibernate will actually be able to choose the correct `org.hibernate.dialect.Dialect` implementation based on the `JDBC metadata` returned by the JDBC driver.
hibernate.show_sql	######### SQL ######################## `org.hibernate.SQL` # `debug` ############ ## `#true` | `false`
hibernate.format_sql	######### SQL ########### `#true` | `false`
hibernate.default_schema	##### SQL ###############/########### `#.SCHEMA_NAME`
hibernate.default_catalog	##### SQL ################ `#CATALOG_NAME`
hibernate.session_factory_name	`org.hibernate.SessionFactory` ######### # JNDI ############ `#jndi/composite/name`
hibernate.max_fetch_depth	################################### ######### `0` ######################

######	##
	## ###### `0` ## `3` #####
hibernate.default_batch_fetch_size	############ Hibernate ############ ## ###### `4` , `8` , `16` ###
hibernate.default_entity_mode	Sets a default mode for entity representation for all sessions opened from this `SessionFactory` `dynamic-map`, `dom4j`, `pojo`
hibernate.order_updates	############## SQL ## ###### #`true` | `false`
hibernate.generate_statistics	###### Hibernate ########################## #`true` | `false`
hibernate.use_identifier_rollback	## #`true` | `false`
hibernate.use_sql_comments	###### SQL ############################## ########## `false` ### #`true` | `false`
hibernate.id.new_generator_mappings	Setting is relevant when using `@GeneratedValue`. It indicates whether or not the new `IdentifierGenerator` implementations are used for `javax.persistence.GenerationType.AUTO`, `javax.persistence.GenerationType.TABLE` and `javax.persistence.GenerationType.SEQUENCE`. Default to `false` to keep backward compatibility. #`true` | `false`

##

We recommend all new projects which make use of to use `@GeneratedValue` to also set `hibernate.id.new_generator_mappings=true` as the new generators

are more efficient and closer to the JPA 2 specification semantic. However they are not backward compatible with existing databases (if a sequence or a table is used for id generation).

#3.4 Hibernate JDBC

######	##
hibernate.jdbc.fetch_size	##0###### JDBC ############ (`Statement.setFetchSize()` #####)#
hibernate.jdbc.batch_size	##0###### Hibernate # JDBC2 ########## ## ## ###### `5` ## `30` #####
hibernate.jdbc.batch_versioned_data	Set this property to `true` if your JDBC driver returns correct row counts from `executeBatch()`. It is usually safe to turn this option on. Hibernate will then use batched DML for automatically versioned data. Defaults to `false`. #`true` | `false`
hibernate.jdbc.factory_class	#### `org.hibernate.jdbc.Batcher` ###### ################################## #`classname.of.BatcherFactory`
hibernate.jdbc.use_scrollable_resultset	Hibernate ### JDBC2 ################## ############################## JDBC # ############################ Hibernate # ################ #`true` | `false`
hibernate.jdbc.use_streams_for_binary	JDBC #/## `binary` # `serializable` #####/## ########### (##############)# #`true` | `false`
hibernate.jdbc.use_get_generated_keys	######################## JDBC3 `PreparedStatement.getGeneratedKeys()` # ############ JDBC3+ ##### JRE1.4+ ## ###### Hibernate ###################### false ################################ ############ #`true`|`false`

######	##
hibernate.connection.provider_class	JDBC ###### Hibernate ######## `ConnectionProvider` ###### `#classname.of.ConnectionProvider`
hibernate.connection.isolation	JDBC ################################ `# java.sql.Connection` ################ ##################################### ## `#1, 2, 4, 8`
hibernate.connection.autocommit	######## JDBC ######################## ### `#true | false`
hibernate.connection.release_mode	Hibernate ### JDBC ################## ################################### ####################### JTA ##### ######### JDBC ################## ######## `after_statement` ######### # JTA ########################## `after_transaction` ################## ### `auto` ##### JTA # CMT ########### `after_statement` ####### JDBC ######### ### `after_transaction` ######### `#auto` (#####) | `on_close` | `after_transaction` | `after_statement` This setting only affects `Sessions` returned from `SessionFactory.openSession`. For `Sessions` obtained through `SessionFactory.getCurrentSession`, the `CurrentSessionContext` implementation configured for use controls the connection release mode for those `Sessions`. See ##########
hibernate.connection.*<propertyName>*	JDBC ##### *<propertyName>* # `DriverManager.getConnection()` ######
hibernate.jndi.*<propertyName>*	##### *<propertyName>* # JNDI `InitialContextFactory` ######

#3.5 Hibernate

######	##
hibernate.cache.provider_class	#### CacheProvider ######## #classname.of.CacheProvider
hibernate.cache.use_minimal_puts	##################################### ##################################### ################ Hibernate3 ########### #################### #true|false
hibernate.cache.use_query_cache	#################################### #true|false
hibernate.cache.use_second_level_cache	##################################### ### <cache> ############ #true|false
hibernate.cache.query_cache_factory	#### QueryCache ###################### ###### StandardQueryCache ###### **e.g.** classname.of.QueryCache
hibernate.cache.region_prefix	################## #prefix
hibernate.cache.use_structured_entries	#################################### #true|false
hibernate.cache.default_cache_concurrency_strategy	Setting used to give the name of the default org.hibernate.annotations.CacheConcurrencyStrategy to use when either @Cacheable or @Cache is used. @Cache(strategy="..") is used to override this default.

#3.6 Hibernate

######	##
hibernate.transaction.factory_class	Hibernate Transaction API ######## TransactionFactory ######## ######## JDBCTransactionFactory #### #classname.of.TransactionFactory >

######	##
jta.UserTransaction	############## JTA UserTransaction ### ##### JTATransactionFactory ##### JNDI # ### #jndi/composite/name
hibernate.transaction.manager_lookup_class	TransactionManagerLookup ######## JTA # ###### JVM ################### hilo ### ################ #classname.of.TransactionManagerLookup
hibernate.transaction.flush_before_completion	If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see ###### #########. #true | false
hibernate.transaction.auto_close_session	If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred, see ###### #########. #true | false

#3.7

######	##
hibernate.current_session_context_class	Supply a custom strategy for the scoping of the "current" Session. See ############### for more information about the built-in strategies. #jta | thread | managed | custom.Class
hibernate.query.factory_class	HQL ############## #org.hibernate.hql.ast.ASTQueryTranslatorFactory or org.hibernate.hql.classic.ClassicQueryTranslatorFactory

hibernate.query.substitutions	HQL # SQL ############### ########### ###########
	
	#hqlLiteral=SQL_LITERAL,
	hqlFunction=SQLFUNC

######	##
`hibernate.hbm2ddl.auto`	`SessionFactory` ################ DDL ################ `create-drop` #### `SessionFactory` ###################### ######## # `validate` | `update` | `create` | `create-drop`
`hibernate.hbm2ddl.import_files`	Comma-separated names of the optional files containing SQL DML statements executed during the `SessionFactory` creation. This is useful for testing or demoing: by adding INSERT statements for example you can populate your database with a minimal set of data when it is deployed. File order matters, the statements of a give file are executed before the statements of the following files. These statements are only executed if the schema is created ie if `hibernate.hbm2ddl.auto` is set to `create` or `create-drop`. **e.g.** `/humans.sql,/dogs.sql`
`hibernate.bytecode.use_reflection_optimizer`	Enables the use of bytecode manipulation instead of runtime reflection. This is a System-level property and cannot be set in `hibernate.cfg.xml`. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either CGLIB or javassist even if you turn off the optimizer. #`true` | `false`
`hibernate.bytecode.provider`	Both javassist or cglib can be used as byte manipulation engines; the default is `javassist`. **e.g.** `javassist` | `cglib`

3.4.1. SQL ###Dialect#

`hibernate.dialect` ###################### `org.hibernate.dialect.Dialect` ######## ###################### Hibernate ## ################

#3.8 Hibernate SQL Dialects (`hibernate.dialect`)

RDBMS	Dialect
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL5	`org.hibernate.dialect.MySQL5Dialect`
MySQL5 with InnoDB	`org.hibernate.dialect.MySQL5InnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle ##############	`org.hibernate.dialect.OracleDialect`
Oracle 9i	`org.hibernate.dialect.Oracle9iDialect`
Oracle 10g	`org.hibernate.dialect.Oracle10gDialect`
Oracle 11g	`org.hibernate.dialect.Oracle10gDialect`
Sybase	`org.hibernate.dialect.SybaseASE15Dialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server 2000	`org.hibernate.dialect.SQLServerDialect`
Microsoft SQL Server 2005	`org.hibernate.dialect.SQLServer2005Dialect`
Microsoft SQL Server 2008	`org.hibernate.dialect.SQLServer2008Dialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
H2 Database	`org.hibernate.dialect.H2Dialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2.

######### ANSI ## Oracle # Sybase ######################## *outer join fetching* ######## # SQL ## ########################1## SQL # `SELECT` ####

`hibernate.max_fetch_depth` ######## `0` ############# ### ############# `1` ########## ############################## `fetch="join"` #############

See ######## for more information.

3.4.3.

Oracle # JDBC ############### `byte` ############# `binary` # `serializable` ########## ######### `hibernate.jdbc.use_streams_for_binary` ########### ### ############### ## #

3.4.4.

The properties prefixed by `hibernate.cache` allow you to use a process or cluster scoped second-level cache system with Hibernate. See the *##2#########* for more information.

3.4.5.

`hibernate.query.substitutions` ########## Hibernate #################

```
hibernate.query.substitutions true=1, false=0
```

####### `true` # `false` ####### SQL #################

```
hibernate.query.substitutions toLowercase=LOWER
```

SQL # `LOWER`

3.4.6. Hibernate

`hibernate.generate_statistics` ################################ `SessionFactory.getStatistics()` ###### Hibernate ############### JMX ########### ########## Javadoc # `org.hibernate.stats` ##############################

3.5.

Hibernate utilizes *Simple Logging Facade for Java* [http://www.slf4j.org/] (SLF4J) in order to log various system events. SLF4J can direct your logging output to several logging frameworks (NOP, Simple, log4j version 1.2, JDK 1.4 logging, JCL or logback) depending on your chosen binding. In order to setup logging you will need `slf4j-api.jar` in your classpath together with the jar file for your preferred binding - `slf4j-log4j12.jar` in the case of Log4J. See the SLF4J *documentation* [http://www.slf4j.org/manual.html] for more detail. To use Log4j you will also need to place a `log4j.properties` file in your classpath. An example properties file is distributed with Hibernate in the `src/` directory.

Hibernate ########################## Hibernate ####################################
##:

#3.9 Hibernate ######

####	##
`org.hibernate.SQL`	######## SQL#DDL#################
`org.hibernate.type`	#### JDBC ##############
`org.hibernate.tool.hbm2ddl`	######## SQL#DDL#################
`org.hibernate.pretty`	session ##
`org.hibernate.cache`	########################
`org.hibernate.transaction`	########################
`org.hibernate.jdbc`	JDBC ###############
`org.hibernate.hql.ast.AST`	HQL # SQL # AST ################
`org.hibernate.secure`	#### JAAS ###########
`org.hibernate`	###

Hibernate ################### `org.hibernate.SQL` ##### `debug` ####################### ####### `hibernate.show_sql` ####################

3.6. `NamingStrategy`

######## `net.sf.hibernate.cfg.NamingStrategy` ################################## ########

Java ## ## `TBL_` ############################ ################ Hibernate ###########################

############ `Configuration.setNamingStrategy()` ###############################

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`org.hibernate.cfg.ImprovedNamingStrategy` ###################################### ###########

3.7. Implementing a PersisterClassProvider

You can configure the persister implementation used to persist your entities and collections:

- by default, Hibernate uses persisters that make sense in a relational model and follow Java Persistence's specification
- you can define a `PersisterClassProvider` implementation that provides the persister class used of a given entity or collection
- finally, you can override them on a per entity and collection basis in the mapping using `@Persister` or its XML equivalent

The latter in the list the higher in priority.

You can pass the `PersisterClassProvider` instance to the `Configuration` object.

```
SessionFactory sf = new Configuration()
    .setPersisterClassProvider(customPersisterClassProvider)
    .addAnnotatedClass(Order.class)
    .buildSessionFactory();
```

The persister class provider methods, when returning a non null persister class, override the default Hibernate persisters. The entity name or the collection role are passed to the methods. It is a nice way to centralize the overriding logic of the persisters instead of spreading them on each entity or collection mapping.

3.8. XML

##1##### `hibernate.cfg.xml` ################################## `hibernate.properties` ##

XML ############ CLASSPATH # root ##################

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
```

```
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

Hibernate ######################## #### `hibernate.cfg.xml` ######## `hibernate.properties` # `hibernate.cfg.xml` # ###### ################2################# XML

XML ######### Hibernate ##################

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

XML

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

3.9. Java EE Application Server integration

Hibernate # J2EE ####################:

- ############## Hibernate # JNDI ############## JDBC ################## JTA ### `TransactionManager` # `ResourceManager` ########### (CMT)######################## ################################### (BMT)################################## Hibernate # `Transaction` API ###############

- *## JNDI* ######## Hibernate # JNDI ######### `SessionFactory` #######

- *JTA* ############# Hibernate `Session` ##### JTA ########################## `SessionFactory` # JNDI ## lookup ###### `Session` ####### JTA ############### ## Hibernate# `Session` ############################### (CMT) ########### (BMT/UserTransaction) ##########

- *JMX #######:* ## JMX ###################### JBoss AS# ###### Hibernate # MBean # ################### `Configuration` ## `SessionFactory` ##########################

HibernateService ################################ Hibernate ##################### ###########

######################### "connection containment" ################ hibernate.connection.aggressive_release # true ########

3.9.1.

Hibernate Session API ## JDBC ########## JDBC API ## ###################### J2EE ##################### Bean ################### UserTransaction # JTA API ##############

2## Hibernate Transaction API ####### Hibernate ######## hibernate.transaction.factory_class ############## Transaction ####################

3####################

org.hibernate.transaction.JDBCTransactionFactory
    ###### (JDBC) ####################

org.hibernate.transaction.JTATransactionFactory
    ################ EJB ##### Bean ### ################################### Bean #################

org.hibernate.transaction.CMTTransactionFactory
    ###### JTA ##############

################### CORBA ##########################

Hibernate ##################### JTA ########################### JTA TransactionManager ############ J2EE ## Hibernate# TransactionManager ############################

#3.10 JTA

Transaction Factory	Application Server
org.hibernate.transaction.JBossTransactionManagerLookup	JBoss AS
org.hibernate.transaction.WeblogicTransactionManagerLookup	Weblogic
org.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
org.hibernate.transaction.WebSphereExtendedJTATransactionLookup	WebSphere 6
org.hibernate.transaction.OrionTransactionManagerLookup	Orion
org.hibernate.transaction.ResinTransactionManagerLookup	Resin
org.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
org.hibernate.transaction.JOnASTransactionManagerLookup	JOnAS

Transaction Factory	Application Server
`org.hibernate.transaction.JRun4TransactionManagerLookup`	JRun4
`org.hibernate.transaction.BESTransactionManagerLookup`	Borland ES
`org.hibernate.transaction.JBossTSStandaloneTransactionManagerLookup`	JBoss TS used standalone (ie. outside JBoss AS and a JNDI environment generally). Known to work for `org.jboss.jbossts:jbossjta:4.11.0.Final`

3.9.2. `SessionFactory` # JNDI

JNDI ##### Hibernate `SessionFactory` ################# `Session` ############## JNDI ###### `Datasource` ################################

`SessionFactory` # JNDI ########################## `java:hibernate/SessionFactory` ## `hibernate.session_factory_name` ############################## ## `SessionFactory` # JNDI ############## Tomcat ############ JNDI ################

`SessionFactory` # JNDI ######## Hibernate # `hibernate.jndi.url` ##### ##`hibernate.jndi.class` ### `InitialContext` #######

`cfg.buildSessionFactory()` ##### Hibernate ##### `SessionFactory` # JNDI ####### `HibernateService` #### JMX ## ############# ##############################

JNDI `SessionFactory` ###### EJB ####### JNDI ########## `SessionFactory`

It is recommended that you bind the `SessionFactory` to JNDI in a managed environment and use a `static` singleton otherwise. To shield your application code from these details, we also recommend to hide the actual lookup code for a `SessionFactory` in a helper class, such as `HibernateUtil.getSessionFactory()`. Note that such a class is also a convenient way to startup Hibernate—see chapter 1.

3.9.3. JTA

The easiest way to handle `Sessions` and transactions is Hibernate's automatic "current" `Session` management. For a discussion of contextual sessions see ###############. Using the `"jta"` session context, if there is no Hibernate `Session` associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call `sessionFactory.getCurrentSession()`. The `Sessions` retrieved via `getCurrentSession()` in the `"jta"` context are set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the `Sessions` to be managed by the life cycle of the JTA transaction to which it

is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through `UserTransaction`, or (recommended for portable code) use the Hibernate `Transaction` API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

3.9.4. JMX

`SessionFactory` # JNDI ########## `cfg.buildSessionFactory()` ######################## ######## `static` ######### `HibernateUtil` ###### *managed service* ### Hibernate ###### ################

JBoss AS #### JMX ########################## `org.hibernate.jmx.HibernateService` ### JBoss 4.0.x ## `jboss-service.xml` ######

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">

    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>
```

```
</server>
```

######## `META-INF` ############ JAR ######### `.sar` (service archive) ############### Hibernate ############### Hibernate ## ############.sar#################### Bean ######### Bean ##### JAR ############1## ## EJB JAR ################ JBoss AS ###### ## JXM ##### EJB ######################

#####

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). The term "persistent" here means that the classes are able to be persisted, not that they are in the persistent state (see *#Hibernate # ############* for discussion).

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of `java.util.Map` instances, for example).

4.1. ### POJO

#4.1 Simple POJO representing a cat

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
private Long id; // identifier

private Date birthdate;
private Color color;
private char sex;
private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }
```

```
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

The four main rules of persistent classes are explored in more detail in the following sections.

4.1.1.

`Cat` has a no-argument constructor. All persistent classes must have a default constructor (which can be non-public) so that Hibernate can instantiate them using `java.lang.reflect.Constructor.newInstance()`. It is recommended that this constructor be defined with at least *package* visibility in order for runtime proxy generation to work properly.

4.1.2. Provide an identifier property

##

Historically this was considered option. While still not (yet) enforced, this should be considered a deprecated feature as it will be completely required to provide a identifier property in an upcoming release.

`Cat` has a property named `id`. This property maps to the primary key column(s) of the underlying database table. The type of the identifier property can be any "basic" type (see *???*). See ##### ############ for information on mapping composite (multi-column) identifiers.

##

Identifiers do not necessarily need to identify column(s) in the database physically defined as a primary key. They should just identify columns that can be used to uniquely identify rows in the underlying table.

null #####(##############)############

4.1.3. Prefer non-final classes (semi-optional)

A central feature of Hibernate, *proxies* (lazy loading), depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods. You can persist `final` classes that do not implement an interface with Hibernate; you will not, however, be able to use proxies for lazy association fetching which will ultimately limit your options for performance tuning. To persist a `final` class which does not implement a "full" interface you must disable proxy generation. See *#4.2#Disabling proxies in hbm.xml#* and *#4.3#Disabling proxies in annotations#*.

#4.2 Disabling proxies in `hbm.xml`

```
<class name="Cat" lazy="false"...>...</class>
```

#4.3 Disabling proxies in annotations

```
@Entity @Proxy(lazy=false) public class Cat { ... }
```

If the `final` class does implement a proper interface, you could alternatively tell Hibernate to use the interface instead when generating the proxies. See *#4.4#Proxying an interface in hbm.xml#* and *#4.5#Proxying an interface in annotations#*.

#4.4 Proxying an interface in `hbm.xml`

```
<class name="Cat" proxy="ICat"...>...</class>
```

#4.5 Proxying an interface in annotations

```
@Entity @Proxy(proxyClass=ICat.class) public class Cat implements ICat { ... }
```

You should also avoid declaring `public final` methods as this will again limit the ability to generate *proxies* from this class. If you want to use a class with `public final` methods, you must explicitly disable proxying. Again, see *#4.2#Disabling proxies in hbm.xml#* and *#4.3#Disabling proxies in annotations#*.

4.1.4.

`Cat` declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. It is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties and recognizes method names of the form `getFoo`, `isFoo` and `setFoo`. If required, you can switch to direct field access for particular properties.

Properties need *not* be declared public. Hibernate can persist a property declared with `package`, `protected` or `private` visibility as well.

4.2.

######1###2################################## `Cat` ##################

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. equals() # hashCode()###

######### equals() # hashCode() ########################

- ############## Set ###### ######################### ######
- ########################

Hibernate #### ID ################################ Java ID ################### ################################ Set ################################ equals() # hashCode() ###############

################################ equals()# hashCode() ########################## ######################### #### Set ######## Set ##1################ ############ ######################### Hibernate ## ## Set ############################## ########### equals() # hashCode() ############################## Set ########### ################ Hibernate ################### Hibernate ############################ ##### Java ######################

########## ##### equals() # hashCode() ############################## equals() #### ###################################### ##### ##########################

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

A business key does not have to be as solid as a database primary key candidate (see ######## ########). Immutable or unique properties are usually good candidates for a business key.

4.4.

##

The following features are currently considered experimental and may change in the near future.

################## POJO #### JavaBean ################## Hibernate ###### `Map` # `Map` ########## DOM4J ## ##########

By default, Hibernate works in normal POJO mode. You can set a default entity representation mode for a particular `SessionFactory` using the `default_entity_mode` configuration option (see *#3.3#Hibernate ########*).

`Map` ## `entity-name` #########

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

POJO

SessionFactory #################### dynamic-map ########### Map # Map ###########

```
Session s = openSession();
Transaction tx = s.beginTransaction();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

Hibernate #########################

############ Session ###############

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

EntityMode #### getSession() ###### SessionFactory #### Session API############# ############ Session ######## JDBC ##2### Session ## flush() # close() ######################################1#######(Unit of Work)##################

More information about the XML representation capabilities can be found in *20#XML #####*.

4.5. Tuplizer

`org.hibernate.tuple.Tuplizer` and its sub-interfaces are responsible for managing a particular representation of a piece of data given that representation's `org.hibernate.EntityMode`. If a given piece of data is thought of as a data structure, then a tuplizer is the thing that knows how to create such a data structure, how to extract values from such a data structure and how to inject values into such a data structure. For example, for the POJO entity mode, the corresponding tuplizer knows how create the POJO through its constructor. It also knows how to access the POJO properties using the defined property accessors.

There are two (high-level) types of Tuplizers:

- `org.hibernate.tuple.entity.EntityTuplizer` which is responsible for managing the above mentioned contracts in regards to entities
- `org.hibernate.tuple.component.ComponentTuplizer` which does the same for components

Users can also plug in their own tuplizers. Perhaps you require that `java.util.Map` implementation other than `java.util.HashMap` be used while in the dynamic-map entity-mode. Or perhaps you need to define a different proxy generation strategy than the one used by default. Both would be achieved by defining a custom tuplizer implementation. Tuplizer definitions are attached to the entity or component mapping they are meant to manage. Going back to the example of our `Customer` entity, *#4.6#Specify custom tuplizers in annotations#* shows how to specify a custom `org.hibernate.tuple.entity.EntityTuplizer` using annotations while *#4.7#Specify custom tuplizers in hbm.xml#* shows how to do the same in `hbm.xml`

#4.6 Specify custom tuplizers in annotations

```
@Entity
@Tuplizer(impl = DynamicEntityTuplizer.class)
public interface Cuisine {
    @Id
    @GeneratedValue
    public Long getId();
    public void setId(Long id);

    public String getName();
    public void setName(String name);

    @Tuplizer(impl = DynamicComponentTuplizer.class)
    public Country getCountry();
    public void setCountry(Country country);
}
```

#4.7 Specify custom tuplizers in `hbm.xml`

```
<hibernate-mapping>
```

```
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>
```

4.6. EntityNameResolvers

`org.hibernate.EntityNameResolver` is a contract for resolving the entity name of a given entity instance. The interface defines a single method `resolveEntityName` which is passed the entity instance and is expected to return the appropriate entity name (null is allowed and would indicate that the resolver does not know how to resolve the entity name of the given entity instance). Generally speaking, an `org.hibernate.EntityNameResolver` is going to be most useful in the case of dynamic models. One example might be using proxied interfaces as your domain model. The hibernate test suite has an example of this exact style of usage under the org.hibernate.test.dynamicentity.tuplizer2. Here is some of the code from that package for illustration.

```
/**
 * A very trivial JDK Proxy InvocationHandler implementation where we proxy an
 * interface as the domain model and simply store persistent state in an internal
 * Map.  This is an extremely trivial example meant only for illustration.
 */
public final class DataProxyHandler implements InvocationHandler {
        private String entityName;
        private HashMap data = new HashMap();

        public DataProxyHandler(String entityName, Serializable id) {
                this.entityName = entityName;
                data.put( "Id", id );
        }

        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                String methodName = method.getName();
                if ( methodName.startsWith( "set" ) ) {
                        String propertyName = methodName.substring( 3 );
                        data.put( propertyName, args[0] );
                }
                else if ( methodName.startsWith( "get" ) ) {
                        String propertyName = methodName.substring( 3 );
                        return data.get( propertyName );
                }
```

```
                else if ( "toString".equals( methodName ) ) {
                        return entityName + "#" + data.get( "Id" );
                }
                else if ( "hashCode".equals( methodName ) ) {
                        return new Integer( this.hashCode() );
                }
                return null;
        }

        public String getEntityName() {
                return entityName;
        }

        public HashMap getData() {
                return data;
        }
}

public class ProxyHelper {
    public static String extractEntityName(Object object) {
        // Our custom java.lang.reflect.Proxy instances actually bundle
        // their appropriate entity name, so we simply extract it from there
        // if this represents one of our proxies; otherwise, we return null
        if ( Proxy.isProxyClass( object.getClass() ) ) {
            InvocationHandler handler = Proxy.getInvocationHandler( object );
            if ( DataProxyHandler.class.isAssignableFrom( handler.getClass() ) ) {
                DataProxyHandler myHandler = ( DataProxyHandler ) handler;
                return myHandler.getEntityName();
            }
        }
        return null;
    }

    // various other utility methods ....

}

/**
 * The EntityNameResolver implementation.
 *
 * IMPL NOTE : An EntityNameResolver really defines a strategy for how entity names
 * should be resolved.  Since this particular impl can handle resolution for all of our
 * entities we want to take advantage of the fact that SessionFactoryImpl keeps these
 * in a Set so that we only ever have one instance registered.  Why?  Well, when it
 * comes time to resolve an entity name, Hibernate must iterate over all the registered
 * resolvers.  So keeping that number down helps that process be as speedy as possible.
 * Hence the equals and hashCode implementations as is
 */
public class MyEntityNameResolver implements EntityNameResolver {
    public static final MyEntityNameResolver INSTANCE = new MyEntityNameResolver();

    public String resolveEntityName(Object entity) {
        return ProxyHelper.extractEntityName( entity );
    }

    public boolean equals(Object obj) {
        return getClass().equals( obj.getClass() );
    }
```

```
    public int hashCode() {
        return getClass().hashCode();
    }
}

public class MyEntityTuplizer extends PojoEntityTuplizer {
       public MyEntityTuplizer(EntityMetamodel entityMetamodel, PersistentClass mappedEntity) {
                super( entityMetamodel, mappedEntity );
        }

        public EntityNameResolver[] getEntityNameResolvers() {
                return new EntityNameResolver[] { MyEntityNameResolver.INSTANCE };
        }

    public String determineConcreteSubclassEntityName(Object entityInstance, SessionFactoryImplementor factory) {
        String entityName = ProxyHelper.extractEntityName( entityInstance );
        if ( entityName == null ) {
            entityName = super.determineConcreteSubclassEntityName( entityInstance, factory );
        }
        return entityName;
    }

    ...
```

In order to register an `org.hibernate.EntityNameResolver` users must either:

1. Implement a custom tuplizer (see *#Tuplizer#*), implementing the `getEntityNameResolvers` method
2. Register it with the `org.hibernate.impl.SessionFactoryImpl` (which is the implementation class for `org.hibernate.SessionFactory`) using the `registerEntityNameResolver` method.

O/R

5.1.

Object/relational mappings can be defined in three approaches:

- using Java 5 annotations (via the Java Persistence 2 annotations)
- using JPA 2 XML deployment descriptors (described in chapter XXX)
- using the Hibernate legacy XML files approach known as hbm.xml

Annotations are split in two categories, the logical mapping annotations (describing the object model, the association between two entities etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc). We will mix annotations from both categories in the following code examples.

JPA annotations are in the `javax.persistence.*` package. Hibernate specific extensions are in `org.hibernate.annotations.*`. You favorite IDE can auto-complete annotations and their attributes for you (even without a specific "JPA" plugin, since JPA annotations are plain Java 5 annotations).

Here is an example of mapping

```
package eg;

@Entity
@Table(name="cats") @Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorValue("C") @DiscriminatorColumn(name="subclass", discriminatorType=CHAR)
public class Cat {

   @Id @GeneratedValue
   public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public BigDecimal getWeight() { return weight; }
   public void setWeight(BigDecimal weight) { this.weight = weight; }
   private BigDecimal weight;

   @Temporal(DATE) @NotNull @Column(updatable=false)
   public Date getBirthdate() { return birthdate; }
   public void setBirthdate(Date birthdate) { this.birthdate = birthdate; }
   private Date birthdate;

   @org.hibernate.annotations.Type(type="eg.types.ColorUserType")
   @NotNull @Column(updatable=false)
   public ColorType getColor() { return color; }
   public void setColor(ColorType color) { this.color = color; }
   private ColorType color;

   @NotNull @Column(updatable=false)
```

```
   public String getSex() { return sex; }
   public void setSex(String sex) { this.sex = sex; }
   private String sex;

   @NotNull @Column(updatable=false)
   public Integer getLitterId() { return litterId; }
   public void setLitterId(Integer litterId) { this.litterId = litterId; }
   private Integer litterId;

   @ManyToOne @JoinColumn(name="mother_id", updatable=false)
   public Cat getMother() { return mother; }
   public void setMother(Cat mother) { this.mother = mother; }
   private Cat mother;

   @OneToMany(mappedBy="mother") @OrderBy("litterId")
   public Set<Cat> getKittens() { return kittens; }
   public void setKittens(Set<Cat> kittens) { this.kittens = kittens; }
   private Set<Cat> kittens = new HashSet<Cat>();
}

@Entity @DiscriminatorValue("D")
public class DomesticCat extends Cat {

   public String getName() { return name; }
   public void setName(String name) { this.name = name }
   private String name;
}

@Entity
public class Dog { ... }
```

The legacy hbm.xml approach uses an XML schema designed to be readable and hand-editable. The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations and not table declarations.

Hibernate ##### XML ################### XDoclet, Middlegen, AndroMDA
##

####################

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>
```

```
                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

We will now discuss the concepts of the mapping documents (both annotations and XML). We will only describe, however, the document elements and attributes that are used by Hibernate at runtime. The mapping document also contains some extra optional attributes and elements that affect the database schemas exported by the schema export tool (for example, the `not-null` attribute).

5.1.1. Entity

An entity is a regular Java object (aka POJO) which will be persisted by Hibernate.

To mark an object as an entity in annotations, use the `@Entity` annotation.

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }
}
```

That's pretty much it, the rest is optional. There are however any options to tweak your entity mapping, let's explore them.

`@Table` lets you define the table the entity will be persisted into. If undefined, the table name is the unqualified class name of the entity. You can also optionally define the catalog, the schema as well as unique constraints on the table.

```
@Entity
@Table(name="TBL_FLIGHT",
       schema="AIR_COMMAND",
       uniqueConstraints=
           @UniqueConstraint(
               name="flight_number",
               columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {
    @Column(name="comp_prefix")
    public String getCompagnyPrefix() { return companyPrefix; }

    @Column(name="flight_number")
    public String getNumber() { return number; }
}
```

The constraint name is optional (generated if left undefined). The column names composing the constraint correspond to the column names as defined before the Hibernate `NamingStrategy` is applied.

`@Entity.name` lets you define the shortcut name of the entity you can used in JP-QL and HQL queries. It defaults to the unqualified class name of the class.

Hibernate goes beyond the JPA specification and provide additional configurations. Some of them are hosted on `@org.hibernate.annotations.Entity`:

- `dynamicInsert` / `dynamicUpdate` (defaults to false): specifies that `INSERT` / `UPDATE` SQL should be generated at runtime and contain only the columns whose values are not null. The `dynamic-`

`update` and `dynamic-insert` settings are not inherited by subclasses. Although these settings can increase performance in some cases, they can actually decrease performance in others.

- `selectBeforeUpdate` (defaults to false): specifies that Hibernate should *never* perform an SQL `UPDATE` unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using `update()`, will Hibernate perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required. Use of `select-before-update` will usually decrease performance. It is useful to prevent a database update trigger being called unnecessarily if you reattach a graph of detached instances to a `Session`.

- `polymorphisms` (defaults to `IMPLICIT`): determines whether implicit or explicit query polymorphisms is used. *Implicit* polymorphisms means that instances of the class will be returned by a query that names any superclass or implemented interface or class, and that instances of any subclass of the class will be returned by a query that names the class itself. *Explicit* polymorphisms means that class instances will be returned only by queries that explicitly name that class. Queries that name the class will return only instances of subclasses mapped. For most purposes, the default `polymorphisms=IMPLICIT` is appropriate. Explicit polymorphisms is useful when two different classes are mapped to the same table This allows a "lightweight" class that contains a subset of the table columns.

- `persister`: specifies a custom `ClassPersister`. The `persister` attribute lets you customize the persistence strategy used for the class. You can, for example, specify your own subclass of `org.hibernate.persister.EntityPersister`, or you can even provide a completely new implementation of the interface `org.hibernate.persister.ClassPersister` that implements, for example, persistence via stored procedure calls, serialization to flat files or LDAP. See `org.hibernate.test.CustomPersister` for a simple example of "persistence" to a `Hashtable`.

- `optimisticLock` (defaults to `VERSION`): determines the optimistic locking strategy. If you enable `dynamicUpdate`, you will have a choice of optimistic locking strategies:

  - `version` #####/####################
  - `all` ################
  - `dirty` ############################
  - `none` #############

  Hibernate ####################/################ ### ########################## ######################## #### `Session.marge()` ######## ######################## ##

Be sure to import `@javax.persistence.Entity` to mark a class as an entity. It's a common mistake to import `@org.hibernate.annotations.Entity` by accident.

Some entities are not mutable. They cannot be updated or deleted by the application. This allows Hibernate to make some minor performance optimizations.. Use the `@Immutable` annotation.

You can also alter how Hibernate deals with lazy initialization for this class. On `@Proxy`, use `lazy`=false to disable lazy fetching (not recommended). You can also specify an interface to use for lazy initializing proxies (defaults to the class itself): use `proxyClass` on `@Proxy`. Hibernate will initially return proxies (Javassist or CGLIB) that implement the named interface. The persistent object will load when a method of the proxy is invoked. See "Initializing collections and proxies" below.

`@BatchSize` specifies a "batch size" for fetching instances of this class by identifier. Not yet loaded instances are loaded batch-size at a time (default 1).

You can specific an arbitrary SQL WHERE condition to be used when retrieving objects of this class. Use `@Where` for that.

In the same vein, `@Check` lets you define an SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

There is no difference between a view and a base table for a Hibernate mapping. This is transparent at the database level, although some DBMS do not support views properly, especially with updates. Sometimes you want to use a view, but you cannot create one in the database (i.e. with a legacy schema). In this case, you can map an immutable and read-only entity to a given SQL subselect expression using `@org.hibernate.annotations.Subselect`:

```
@Entity
@Subselect("select item.name, max(bid.amount), count(*) "
        + "from item "
        + "join bid on bid.item_id = item.id "
        + "group by item.name")
@Synchronize( {"item", "bid"} ) //tables impacted
public class Summary {
    @Id
    public String getId() { return id; }
    ...
}
```

`<subselect>`

We will now explore the same options using the hbm.xml structure. You can declare a persistent class using the `class` element. For example:

```
<class
        name="ClassName"                                          (1)
        table="tableName"                                         (2)
        discriminator-value="discriminator_value"                 (3)
        mutable="true|false"                                      (4)
```

```
        schema="owner"                                                  ❺
        catalog="catalog"                                               ❻
        proxy="ProxyInterface"                                          ❼
        dynamic-update="true|false"                                     ❽
        dynamic-insert="true|false"                                     ❾
        select-before-update="true|false"                               ❿
        polymorphism="implicit|explicit"                                ⓫
        where="arbitrary sql where condition"                           ⓬
        persister="PersisterClass"                                      ⓭
        batch-size="N"                                                  ⓮
        optimistic-lock="none|version|dirty|all"                        ⓯
        lazy="true|false"                                               (16)
        entity-name="EntityName"                                        (17)
        check="arbitrary sql check condition"                           (18)
        rowid="rowid"                                                   (19)
        subselect="SQL expression"                                      (20)
        abstract="true|false"                                           (21)
        node="element-name"
/>
```

❶ `name` (#####)######################### Java ###################### POJO ###### ########################

❷ `table` (##### - ###################)################

❸ `discriminator-value` (##### - ##########)# ################################## #### `null` # `not null` ###########

❹ `mutable` (############ `true`)# ######################################

❺ `schema` (#####): #### <hibernate-mapping> ########################

❻ `catalog` (#####): #### <hibernate-mapping> ########################

❼ `proxy` ##

❽ `dynamic-update` ######## ###### `false` ################## SQL # `UPDATE` ######### ###########

❾ `dynamic-insert` ######, ###### `false` #### null ############ SQL # `INSERT` ###### ##############

❿ `select-before-update` (############ `false`): ######################## Hibernate # SQL # `UPDATE` # ######## ################(################ `update()` ######### ##############)# `UPDATE` ################### Hibernate #### SQL # `SELECT` ##### ##########

⓫ `polymorphisms` (optional - defaults to `implicit`): determines whether implicit or explicit query polymorphisms is used.

⓬ `where` ######## ############################## SQL # `WHERE` #########

⓭ `persister` ############ `ClassPersister` #######

⓮ `batch-size` ############# `1` ## ######################################

⓯ `optimistic-lock` ############# `version` ## ##############

16 `lazy` ######## `lazy="false"` ##########################

17 `entity-name` (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See ####### and *20#XML #####* for more information.

18 `check` ############################## *check* ####### SQL ##

19 `rowid` ######## Hibernate ###################### ROWID # ###################### Oracle ################## `rowid` ####### Hiberante # update ######### `rowid` ###### ############## ROWID ###################################

20 `subselect` (optional): maps an immutable and read-only entity to a database subselect. This is useful if you want to have a view instead of a base table. See below for more information.

21 `abstract` ######## `<union-subclass>` ############################

`<subclass>` ########################## ######### *static* ############################## `eg.Foo$Bar`

Here is how to do a virtual view (subselect) in XML:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

The `<subselect>` is available both as an attribute and a nested mapping element.

5.1.2. Identifiers

Mapped classes *must* declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance.

Mark the identifier property with `@Id`.

```
@Entity
public class Person {
   @Id Integer getId() { ... }
   ...
}
```

In hbm.xml, use the `<id>` element which defines the mapping from that property to the primary key column.

```
<id
        name="propertyName"                                                    (1)
        type="typename"                                                        (2)
        column="column_name"                                                   (3)
        unsaved-value="null|any|none|undefined|id_value"                       (4)
        access="field|property|ClassName">                                     (5)
        node="element-name|@attribute-name|element/@attribute|."

        <generator class="generatorClass"/>
</id>
```

(1) name####################

(2) type######## Hibernate ########

(3) column###### - ############## ##########

(4) unsaved-value###### - ######## sensible ## ##################### ############## ################## Session ###################################

(5) access (##### - ###### property): Hibernate ############################

name ###################################

The unsaved-value attribute is almost never needed in Hibernate3 and indeed has no corresponding element in annotations.

You can also declare the identifier as a composite identifier. This allows access to legacy data with composite keys. Its use is strongly discouraged for anything else.

5.1.2.1. Composite identifier

You can define a composite primary key through several syntaxes:

- use a component type to represent the identifier and map it as a property in the entity: you then annotated the property as @EmbeddedId. The component type has to be Serializable.
- map multiple properties as @Id properties: the identifier type is then the entity class itself and needs to be Serializable. This approach is unfortunately not standard and only supported by Hibernate.
- map multiple properties as @Id properties and declare an external class to be the identifier type. This class, which needs to be Serializable, is declared on the entity via the @IdClass annotation. The identifier type must contain the same properties as the identifier properties of the entity: each property name must be the same, its type must be the same as well if the entity property is of a basic type, its type must be the type of the primary key of the associated entity if the entity property is an association (either a @OneToOne or a @ManyToOne).

As you can see the last case is far from obvious. It has been inherited from the dark ages of EJB 2 for backward compatibilities and we recommend you not to use it (for simplicity sake).

Let's explore all three cases using examples.

5.1.2.1.1. id as a property using a component type

Here is a simple example of `@EmbeddedId`.

```
@Entity
class User {
   @EmbeddedId
   @AttributeOverride(name="firstName", column=@Column(name="fld_firstname")
   UserId id;

   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;
}
```

You can notice that the `UserId` class is serializable. To override the column mapping, use `@AttributeOverride`.

An embedded id can itself contains the primary key of an associated entity.

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;

   @MapsId("userId")
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   @OneToOne User user;
}

@Embeddable
class CustomerId implements Serializable {
   UserId userId;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
```

```
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In the embedded id object, the association is represented as the identifier of the associated entity. But you can link its value to a regular association in the entity via the `@MapsId` annotation. The `@MapsId` value correspond to the property name of the embedded id object containing the associated entity's identifier. In the database, it means that the `Customer.user` and the `CustomerId.userId` properties share the same underlying column (`user_fk` in this case).

####

The component type used as identifier must implement `equals()` and `hashCode()`.

In practice, your code only sets the `Customer.user` property and the user id value is copied by Hibernate into the `CustomerId.userId` property.

##

The id value can be copied as late as flush time, don't rely on it until after flush time.

While not supported in JPA, Hibernate lets you place your association directly in the embedded id component (instead of having to use the `@MapsId` annotation).

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;
}

@Embeddable
class CustomerId implements Serializable {
   @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
```

```
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;


   //implements equals and hashCode
}
```

Let's now rewrite these examples using the hbm.xml syntax.

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName"
        node="element-name|.">

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName" class="ClassName" column="column_name"/>
        ......
</composite-id>
```

First a simple example:

```
<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName" column="fld_firstname"/>
      <key-property name="lastName"/>
   </composite-id>
</class>
```

Then an example showing how an association can be mapped.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-property name="firstName" column="userfirstname_fk"/>
      <key-property name="lastName" column="userfirstname_fk"/>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>

   <many-to-one name="user">
      <column name="userfirstname_fk" updatable="false" insertable="false"/>
      <column name="userlastname_fk" updatable="false" insertable="false"/>
   </many-to-one>
</class>
```

```
<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

Notice a few things in the previous example:

- the order of the properties (and column) matters. It must be the same between the association and the primary key of the associated entity
- the many to one uses the same columns as the primary key and thus must be marked as read only (`insertable` and `updatable` to false).
- unlike with `@MapsId`, the id value of the associated entity is not transparently copied, check the `foreign` id generator for more information.

The last example shows how to map association directly in the embedded id component.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

This is the recommended approach to map composite identifier. The following options should not be considered unless some constraint are present.

5.1.2.1.2. Multiple id properties without identifier type

Another, arguably more natural, approach is to place `@Id` on multiple properties of your entity. This approach is only supported by Hibernate (not JPA compliant) but does not require an extra embeddable component.

```
@Entity
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In this case `Customer` is its own identifier representation: it must implement `Serializable` and must implement `equals()` and `hashCode()`.

In hbm.xml, the same mapping is:

```
<class name="Customer">
   <composite-id>
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
```

```
</class>
```

5.1.2.1.3. Multiple id properties with with a dedicated identifier type

`@IdClass` on an entity points to the class (component) representing the identifier of the class. The properties marked `@Id` on the entity must have their corresponding property on the `@IdClass`. The return type of search twin property must be either identical for basic properties or must correspond to the identifier class of the associated entity for an association.

##

This approach is inherited from the EJB 2 days and we recommend against its use. But, after all it's your application and Hibernate supports it.

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   UserId user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

`Customer` and `CustomerId` do have the same properties `customerNumber` as well as `user`. `CustomerId` must be `Serializable` and implement `equals()` and `hashCode()`.

While not JPA standard, Hibernate let's you declare the vanilla associated property in the `@IdClass`.

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   @OneToOne User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

This feature is of limited interest though as you are likely to have chosen the `@IdClass` approach to stay JPA compliant or you have a quite twisted mind.

Here are the equivalent on hbm.xml files:

```
<class name="Customer">
   <composite-id class="CustomerId" mapped="true">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
```

```
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

5.1.2.2. Identifier generator

Hibernate can generate and populate identifier values for you automatically. This is the recommended approach over "business" or "natural" id (especially composite ids).

Hibernate offers various generation strategies, let's explore the most common ones first that happens to be standardized by JPA:

- IDENTITY: supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type `long`, `short` or `int`.
- SEQUENCE (called `seqhilo` in Hibernate): uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a named database sequence.
- TABLE (called `MultipleHiLoPerTableGenerator` in Hibernate) : uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a table and column as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
- AUTO: selects `IDENTITY`, `SEQUENCE` or `TABLE` depending upon the capabilities of the underlying database.

####

We recommend all new projects to use the new enhanced identifier generators. They are deactivated by default for entities using annotations but can be activated using `hibernate.id.new_generator_mappings=true`. These new generators are more efficient and closer to the JPA 2 specification semantic.

However they are not backward compatible with existing Hibernate based application (if a sequence or a table is used for id generation). See XXXXXXX *???* for more information on how to activate them.

To mark an id property as generated, use the `@GeneratedValue` annotation. You can specify the strategy used (default to `AUTO`) by setting `strategy`.

```
@Entity
public class Customer {
   @Id @GeneratedValue
   Integer getId() { ... };
}

@Entity
public class Invoice {
   @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
   Integer getId() { ... };
}
```

`SEQUENCE` and `TABLE` require additional configurations that you can set using `@SequenceGenerator` and `@TableGenerator`:

- `name`: name of the generator
- `table` / `sequenceName`: name of the table or the sequence (defaulting respectively to `hibernate_sequences` and `hibernate_sequence`)
- `catalog` / `schema`:
- `initialValue`: the value from which the id is to start generating
- `allocationSize`: the amount to increment by when allocating id numbers from the generator

In addition, the `TABLE` strategy also let you customize:

- `pkColumnName`: the column name containing the entity identifier
- `valueColumnName`: the column name containing the identifier value
- `pkColumnValue`: the entity identifier
- `uniqueConstraints`: any potential column constraint on the table containing the ids

To link a table or sequence generator definition with an actual generated property, use the same name in both the definition `name` and the generator value `generator` as shown below.

```
@Id
@GeneratedValue(
    strategy=GenerationType.SEQUENCE,
    generator="SEQ_GEN")
@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
public Integer getId() { ... }
```

The scope of a generator definition can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined in JPA's XML deployment descriptors (see XXXXXX *???*):

```
<table-generator name="EMP_GEN"
            table="GENERATOR_TABLE"
            pk-column-name="key"
            value-column-name="hi"
            pk-column-value="EMP"
            allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi"
    pkColumnValue="EMP",
    allocationSize=20
)

<sequence-generator name="SEQ_GEN"
    sequence-name="my_sequence"
    allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
```

If a JPA XML descriptor (like `META-INF/orm.xml`) is used to define the generators, `EMP_GEN` and `SEQ_GEN` are application level generators.

##

Package level definition is not supported by the JPA specification. However, you can use the `@GenericGenerator` at the package level (see *???*).

These are the four standard JPA generators. Hibernate goes beyond that and provide additional generators or additional options as we will see below. You can also write your own custom identifier generator by implementing `org.hibernate.id.IdentifierGenerator`.

To define a custom generator, use the `@GenericGenerator` annotation (and its plural counter part `@GenericGenerators`) that describes the class of the identifier generator or its short cut name (as described below) and a list of key/value parameters. When using `@GenericGenerator` and

assigning it via `@GeneratedValue.generator`, the `@GeneratedValue.strategy` is ignored: leave it blank.

```
@Id @GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid", strategy = "uuid")
public String getId() {

@Id @GeneratedValue(generator="trigger-generated")
@GenericGenerator(
    name="trigger-generated",
    strategy = "select",
    parameters = @Parameter(name="key", value = "socialSecurityNumber")
)
public String getId() {
```

The hbm.xml approach uses the optional `<generator>` child element inside `<id>`. If any parameters are required to configure or initialize the generator instance, they are passed using the `<param>` element.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

5.1.2.2.1. Various additional generators

`############` `org.hibernate.id.IdentifierGenerator` `############################` `##########################################` Hibernate `########################` `##########################`

`increment`

`long` , `short` , `int` `#####################################################` `#######` `#########` #

`identity`

DB2, MySQL, MS SQL Server, Sybase, HypersonicSQL `##########################` `long` , `short` , `int` `########`

`sequence`

DB2, PostgreSQL, Oracle, SAP DB, McKoi `########` Interbase `######################` ## `long` , `short` , `int` `########`

`hilo`

`long` , `short` , `int` `###############` hi/lo `#############` hi `######################`(`###` `########` `hibernate_unique_key` # `next_hi`)# hi/lo `##############################` `#######`

`seqhilo`

`long` , `short` , `int` ########## hi/lo ##########

`uuid`

Generates a 128-bit UUID based on a custom algorithm. The value generated is represented as a string of 32 hexidecimal digits. Users can also configure it to use a separator (config parameter "separator") which separates the hexidecimal digits into 8{sep}8{sep}4{sep}8{sep}4. Note specifically that this is different than the IETF RFC 4122 representation of 8-4-4-4-12. If you need RFC 4122 compliant UUIDs, consider using "uuid2" generator discussed below.

`uuid2`

Generates a IETF RFC 4122 compliant (variant 2) 128-bit UUID. The exact "version" (the RFC term) generated depends on the pluggable "generation strategy" used (see below). Capable of generating values as `java.util.UUID`, `java.lang.String` or as a byte array of length 16 (`byte[16]`). The "generation strategy" is defined by the interface `org.hibernate.id.UUIDGenerationStrategy`. The generator defines 2 configuration parameters for defining which generation strategy to use:

`uuid_gen_strategy_class`

Names the UUIDGenerationStrategy class to use

`uuid_gen_strategy`

Names the UUIDGenerationStrategy instance to use

Out of the box, comes with the following strategies:

- `org.hibernate.id.uuid.StandardRandomStrategy` (the default) - generates "version 3" (aka, "random") UUID values via the `randomUUID` method of `java.util.UUID`
- `org.hibernate.id.uuid.CustomVersionOneStrategy` - generates "version 1" UUID values, using IP address since mac address not available. If you need mac address to be used, consider leveraging one of the existing third party UUID generators which sniff out mac address and integrating it via the `org.hibernate.id.UUIDGenerationStrategy` contract. Two such libraries known at time of this writing include *http://johannburkard.de/software/uuid/* and *http://commons.apache.org/sandbox/id/uuid.html*

`guid`

MS SQL ##### MySQL ########## GUID ##########

`native`

########## `identity` # `sequence` # `hilo` ##########

`assigned`

`save()` ########## `<generator>` ########## ##########

`select`

##########

foreign
    ######################## <one-to-one> #################

sequence-identity
    ## JDBC3 getGeneratedKeys ####### INSERT ################################## JDK 1.4 ###### Oracle 10g ########## ################### INSERT ######## Oracle ##############################

5.1.2.2.2. Hi/lo

hilo # seqhilo ########################## hi/lo #######2############1########## #### "hi" ##############################2######## Oracle ########################## ####

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Hibernate ##### Connection ########## hilo ##### ## Hibernate # JTA ## hibernate.transaction.manager_lookup_class

5.1.2.2.3. UUID

UUID ############## IP ##### JVM #############4##1################ JVM ########## ###### Java ##### MAC ########################### JNI ################

5.1.2.2.4.

########################DB2, MySQL, Sybase, MS SQL#### identity ################ ##############DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB#### sequence ######## ################################### SQL ####2########

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
```

```
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

##################native ### identity # sequence # hilo ######1##################
#################

5.1.2.2.5.

If you want the application to assign identifiers, as opposed to having Hibernate generate them, you can use the `assigned` generator. This special generator uses the identifier value already assigned to the object's identifier property. The generator is used when the primary key is a natural key instead of a surrogate key. This is the default behavior if you do not specify `@GeneratedValue` nor `<generator>` elements.

`assigned` ############## Hibernate # `unsaved-value="undefined"` ##################### ################ `Interceptor.isUnsaved()` ########################(transient)####### ###############(detached)#############################

5.1.2.2.6.

####################(Hibernate ######## DDL #######)#

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

########################## `socialSecurityNumber` ################################## # `person_id` ################

5.1.2.2.7. Identity copy (foreign generator)

Finally, you can ask Hibernate to copy the identifier from another associated entity. In the Hibernate jargon, it is known as a foreign generator but the JPA mapping reads better and is encouraged.

```
@Entity
class MedicalHistory implements Serializable {
  @Id @OneToOne
  @JoinColumn(name = "person_id")
  Person patient;
}
```

```
@Entity
public class Person implements Serializable {
  @Id @GeneratedValue Integer id;
}
```

Or alternatively

```
@Entity
class MedicalHistory implements Serializable {
  @Id Integer id;

  @MapsId @OneToOne
  @JoinColumn(name = "patient_id")
  Person patient;
}

@Entity
class Person {
  @Id @GeneratedValue Integer id;
}
```

In hbm.xml use the following approach:

```
<class name="MedicalHistory">
   <id name="id">
      <generator class="foreign">
         <param name="property">patient</param>
      </generator>
   </id>
   <one-to-one name="patient" class="Person" constrained="true"/>
</class>
```

5.1.2.3. Enhanced identifier generators

Starting with release 3.2.3, there are 2 new generators which represent a re-thinking of 2 different aspects of identifier generation. The first aspect is database portability; the second is optimization Optimization means that you do not have to query the database for every request for a new identifier value. These two new generators are intended to take the place of some of the named generators described above, starting in 3.3.x. However, they are included in the current releases and can be referenced by FQN.

The first of these new generators is `org.hibernate.id.enhanced.SequenceStyleGenerator` which is intended, firstly, as a replacement for the `sequence` generator and, secondly, as a better portability generator than `native`. This is because `native` generally chooses between `identity` and `sequence` which have largely different semantics that can cause subtle issues in applications eyeing portability. `org.hibernate.id.enhanced.SequenceStyleGenerator`, however, achieves portability in a different manner. It chooses between a table or a sequence in the database to store its incrementing values, depending on the capabilities of the dialect being used. The difference

between this and `native` is that table-based and sequence-based storage have the same exact semantic. In fact, sequences are exactly what Hibernate tries to emulate with its table-based generators. This generator has a number of configuration parameters:

- `sequence_name` (optional, defaults to `hibernate_sequence`): the name of the sequence or table to be used.
- `initial_value` (optional, defaults to `1`): the initial value to be retrieved from the sequence/table. In sequence creation terms, this is analogous to the clause typically named "STARTS WITH".
- `increment_size` (optional - defaults to `1`): the value by which subsequent calls to the sequence/table should differ. In sequence creation terms, this is analogous to the clause typically named "INCREMENT BY".
- `force_table_use` (optional - defaults to `false`): should we force the use of a table as the backing structure even though the dialect might support sequence?
- `value_column` (optional - defaults to `next_val`): only relevant for table structures, it is the name of the column on the table which is used to hold the value.
- `optimizer` (optional - defaults to `none`): See *#Identifier generator optimization#*

The second of these new generators is `org.hibernate.id.enhanced.TableGenerator`, which is intended, firstly, as a replacement for the `table` generator, even though it actually functions much more like `org.hibernate.id.MultipleHiLoPerTableGenerator`, and secondly, as a re-implementation of `org.hibernate.id.MultipleHiLoPerTableGenerator` that utilizes the notion of pluggable optimizers. Essentially this generator defines a table capable of holding a number of different increment values simultaneously by using multiple distinctly keyed rows. This generator has a number of configuration parameters:

- `table_name` (optional - defaults to `hibernate_sequences`): the name of the table to be used.
- `value_column_name` (optional - defaults to `next_val`): the name of the column on the table that is used to hold the value.
- `segment_column_name` (optional - defaults to `sequence_name`): the name of the column on the table that is used to hold the "segment key". This is the value which identifies which increment value to use.
- `segment_value` (optional - defaults to `default`): The "segment key" value for the segment from which we want to pull increment values for this generator.
- `segment_value_length` (optional - defaults to `255`): Used for schema generation; the column size to create this segment key column.
- `initial_value` (optional - defaults to `1`): The initial value to be retrieved from the table.
- `increment_size` (optional - defaults to `1`): The value by which subsequent calls to the table should differ.
- `optimizer` (optional - defaults to `??`): See *#Identifier generator optimization#*.

5.1.2.3.1. Identifier generator optimization

For identifier generators that store values in the database, it is inefficient for them to hit the database on each and every call to generate a new identifier value. Instead, you can group a bunch of them in memory and only hit the database when you have exhausted your in-memory value

group. This is the role of the pluggable optimizers. Currently only the two enhanced generators (*#Enhanced identifier generators#* support this operation.

- `none` (generally this is the default if no optimizer was specified): this will not perform any optimizations and hit the database for each and every request.
- `hilo`: applies a hi/lo algorithm around the database retrieved values. The values from the database for this optimizer are expected to be sequential. The values retrieved from the database structure for this optimizer indicates the "group number". The `increment_size` is multiplied by that value in memory to define a group "hi value".
- `pooled`: as with the case of `hilo`, this optimizer attempts to minimize the number of hits to the database. Here, however, we simply store the starting value for the "next group" into the database structure rather than a sequential value in combination with an in-memory grouping algorithm. Here, `increment_size` refers to the values coming from the database.

5.1.2.4. Partial identifier generation

Hibernate supports the automatic generation of some of the identifier properties. Simply use the `@GeneratedValue` annotation on one or several id properties.

##

The Hibernate team has always felt such a construct as fundamentally wrong. Try hard to fix your data model before using this feature.

```
@Entity
public class CustomerInventory implements Serializable {
  @Id
  @TableGenerator(name = "inventory",
    table = "U_SEQUENCES",
    pkColumnName = "S_ID",
    valueColumnName = "S_NEXTNUM",
    pkColumnValue = "inventory",
    allocationSize = 1000)
  @GeneratedValue(strategy = GenerationType.TABLE, generator = "inventory")
  Integer id;


  @Id @ManyToOne(cascade = CascadeType.MERGE)
  Customer customer;
}

@Entity
public class Customer implements Serializable {
   @Id
   private int id;
}
```

You can also generate properties inside an `@EmbeddedId` class.

5.1.3. Optimistic locking properties (optional)

When using long transactions or conversations that span several database transactions, it is useful to store versioning data to ensure that if the same entity is updated by two conversations, the last to commit changes will be informed and not override the other conversation's work. It guarantees some isolation while still allowing for good scalability and works particularly well in read-often write-sometimes situations.

You can use two approaches: a dedicated version number or a timestamp.

A version or timestamp property should never be null for a detached instance. Hibernate will detect any instance with a null version or timestamp as transient, irrespective of what other `unsaved-value` strategies are specified. *Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate. It is especially useful for people using assigned identifiers or composite keys*.

5.1.3.1. Version number

You can add optimistic locking capability to an entity using the `@Version` annotation:

```
@Entity
public class Flight implements Serializable {
...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}
```

The version property will be mapped to the `OPTLOCK` column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

The version column may be a numeric. Hibernate supports any kind of type provided that you define and implement the appropriate `UserVersionType`.

The application must not alter the version number set up by Hibernate in any way. To artificially increase the version number, check in Hibernate Entity Manager's reference documentation `LockModeType.OPTIMISTIC_FORCE_INCREMENT` or `LockModeType.PESSIMISTIC_FORCE_INCREMENT`.

If the version number is generated by the database (via a trigger for example), make sure to use `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

To declare a version property in hbm.xml, use:

```
<version
        column="version_column"                    (1)
        name="propertyName"                        (2)
```

```
        type="typename"                                                    3
        access="field|property|ClassName"                                  4
        unsaved-value="null|negative|undefined"                            5
        generated="never|always"                                           6
        insert="true|false"                                                7
        node="element-name|@attribute-name|element/@attribute|."
/>
```

1. `column` ###### - ##############: ####################
2. `name` #################
3. `type` ###### - ###### `integer` #############
4. `access` (##### - ###### `property`): Hibernate ############################
5. `unsaved-value` (optional - defaults to `undefined`): a version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session. `Undefined` specifies that the identifier property value should be used.
6. `generated` (optional - defaults to `never`): specifies that this version property value is generated by the database. See the discussion of *generated properties* for more information.
7. `insert` (##### - ###### `true`): SQL# insert ## ########## `0` ########### `false` ##############

5.1.3.2. Timestamp

Alternatively, you can use a timestamp. Timestamps are a less safe implementation of optimistic locking. However, sometimes an application might use the timestamps in other ways as well.

Simply mark a property of type `Date` or `Calendar` as `@Version`.

```
@Entity
public class Flight implements Serializable {
...
    @Version
    public Date getLastUpdate() { ... }
}
```

When using timestamp versioning you can tell Hibernate where to retrieve the timestamp value from - database or JVM - by optionally adding the `@org.hibernate.annotations.Source` annotation to the property. Possible values for the value attribute of the annotation are `org.hibernate.annotations.SourceType.VM` and `org.hibernate.annotations.SourceType.DB`. The default is `SourceType.DB` which is also used in case there is no `@Source` annotation at all.

Like in the case of version numbers, the timestamp can also be generated by the database instead of Hibernate. To do that, use `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

In hbm.xml, use the <timestamp> element:

```
<timestamp
        column="timestamp_column"
        name="propertyName"
        access="field|property|ClassName"
        unsaved-value="null|undefined"
        source="vm|db"
        generated="never|always"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

1. `column`###### - ############### ####################
2. `name` # ######## Java # `Date`#### `Timestamp` # ## JavaBeans #############
3. `access` (###### - ####### `property`): Hibernate ############################
4. `unsaved-value` ###### - ###### `null` ## ###################### #################### ############### Session ### # `undefined` ## #########################
5. `source` (###### - ####### `vm`): Hibernate ## ###### JVM ############################ Hibernate # "###" ######################### ################################## JVM ### ####### `Dialect` ### ###### (### Oracle 8)#
6. `generated` (optional - defaults to `never`): specifies that this timestamp property value is actually generated by the database. See the discussion of *generated properties* for more information.

> **##**
>
> <timestamp> # <version type="timestamp"> #################### <timestamp source="db"> # <version type="dbtimestamp"> ############# #######

5.1.4. property

You need to decide which property needs to be made persistent in a given entity. This differs slightly between the annotation driven metadata and the hbm.xml files.

5.1.4.1. Property mapping with annotations

In the annotations world, every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as `@Transient`. Not having an annotation for your property is equivalent to the appropriate `@Basic` annotation.

The `@Basic` annotation allows you to declare the fetching strategy for a property. If set to `LAZY`, specifies that this property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation, if your classes are not instrumented, property level lazy loading is silently ignored. The default is `EAGER`. You can also mark a property as not optional thanks to the `@Basic.optional` attribute. This will ensure that the underlying column are not nullable (if possible). Note that a better approach is to use the `@NotNull` annotation of the Bean Validation specification.

Let's look at a few examples:

```
public transient int counter; //transient property

private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property

String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

`counter`, a transient field, and `lengthInMeter`, a method annotated as `@Transient`, and will be ignored by the Hibernate. `name`, `length`, and `firstname` properties are mapped persistent and eagerly fetched (the default for simple properties). The `detailedComment` property value will be lazily fetched from the database once a lazy property of the entity is accessed for the first time. Usually you don't need to lazy simple properties (not to be confused with lazy association fetching). The recommended alternative is to use the projection capability of JP-QL (Java Persistence Query Language) or Criteria queries.

JPA support property mapping of all basic types supported by Hibernate (all basic Java types , their respective wrappers and serializable classes). Hibernate Annotations supports out of the box enum type mapping either into a ordinal column (saving the enum ordinal) or a string based column (saving the enum string representation): the persistence representation, defaulted to ordinal, can be overridden through the `@Enumerated` annotation as shown in the `note` property example.

In plain Java APIs, the temporal precision of time is not defined. When dealing with temporal data you might want to describe the expected precision in database. Temporal data can have `DATE`, `TIME`, or `TIMESTAMP` precision (ie the actual date, only the time, or both). Use the `@Temporal` annotation to fine tune that.

`@Lob` indicates that the property should be persisted in a Blob or a Clob depending on the property type: `java.sql.Clob`, `Character[]`, `char[]` and java.lang.`String` will be persisted in a Clob. `java.sql.Blob`, `Byte[]`, `byte[]` and `Serializable` type will be persisted in a Blob.

```
@Lob
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
}
```

If the property type implements `java.io.Serializable` and is not a basic type, and if the property is not annotated with `@Lob`, then the Hibernate `serializable` type is used.

5.1.4.1.1. Type

You can also manually specify a type using the `@org.hibernate.annotations.Type` and some parameters if needed. `@Type.type` could be:

1. Hibernate ######### `integer, string, character, date, timestamp, float, binary, serializable, object, blob` ##
2. ########## Java #### ## `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob` ##
3. ######### Java #######
4. ############ `com.illflow.type.MyCustomType` ##

If you do not specify a type, Hibernate will use reflection upon the named property and guess the correct Hibernate type. Hibernate will attempt to interpret the name of the return class of the property getter using, in order, rules 2, 3, and 4.

`@org.hibernate.annotations.TypeDef` and `@org.hibernate.annotations.TypeDefs` allows you to declare type definitions. These annotations can be placed at the class or package level. Note that these definitions are global for the session factory (even when defined at the class level). If the type is used on a single entity, you can place the definition on the entity itself. Otherwise, it is recommended to place the definition at the package level. In the example below, when Hibernate encounters a property of class `PhoneNumer`, it delegates the persistence strategy to the custom mapping type `PhoneNumberType`. However, properties belonging to other classes, too, can delegate their persistence strategy to `PhoneNumberType`, by explicitly using the `@Type` annotation.

##

Package level annotations are placed in a file named `package-info.java` in the appropriate package. Place your annotations before the package declaration.

```
@TypeDef(
   name = "phoneNumber",
   defaultForType = PhoneNumber.class,
   typeClass = PhoneNumberType.class
)

@Entity
public class ContactDetails {
   [...]
   private PhoneNumber localPhoneNumber;
   @Type(type="phoneNumber")
   private OverseasPhoneNumber overseasPhoneNumber;
   [...]
}
```

The following example shows the usage of the `parameters` attribute to customize the TypeDef.

```
//in org/hibernate/test/annotations/entity/package-info.java
@TypeDefs(
    {
    @TypeDef(
        name="caster",
        typeClass = CasterStringType.class,
        parameters = {
            @Parameter(name="cast", value="lower")
        }
    )
    }
)
package org.hibernate.test.annotations.entity;

//in org/hibernate/test/annotations/entity/Forest.java
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

When using composite user type, you will have to express column definitions. The `@Columns` has been introduced for that purpose.

```
@Type(type="org.hibernate.test.annotations.entity.MonetaryAmountUserType")
@Columns(columns = {
    @Column(name="r_amount"),
    @Column(name="r_currency")
})
public MonetaryAmount getAmount() {
    return amount;
}


public class MonetaryAmount implements Serializable {
```

```
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

5.1.4.1.2. Access type

By default the access type of a class hierarchy is defined by the position of the `@Id` or `@EmbeddedId` annotations. If these annotations are on a field, then only fields are considered for persistence and the state is accessed via the field. If there annotations are on a getter, then only the getters are considered for persistence and the state is accessed via the getter/setter. That works well in practice and is the recommended approach.

##

The placement of annotations within a class hierarchy has to be consistent (either field or on property) to be able to determine the default access type. It is recommended to stick to one single annotation placement strategy throughout your whole application.

However in some situations, you need to:

- force the access type of the entity hierarchy
- override the access type of a specific entity in the class hierarchy
- override the access type of an embeddable type

The best use case is an embeddable class used by several entities that might not use the same access type. In this case it is better to force the access type at the embeddable class level.

To force the access type on a given class, use the `@Access` annotation as showed below:

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   @Embedded private Address address;
   public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Entity
public class User {
   private Long id;
   @Id public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
```

```
   private Address address;
   @Embedded public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Embeddable
@Access(AcessType.PROPERTY)
public class Address {
   private String street1;
   public String getStreet1() { return street1; }
   public void setStreet1() { this.street1 = street1; }

   private hashCode; //not persistent
}
```

You can also override the access type of a single property while keeping the other properties standard.

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   @Transient private String userId;
   @Transient private String orderId;

   @Access(AccessType.PROPERTY)
   public String getOrderNumber() { return userId + ":" + orderId; }
   public void setOrderNumber() { this.userId = ...; this.orderId = ...; }
}
```

In this example, the default access type is `FIELD` except for the `orderNumber` property. Note that the corresponding field, if any must be marked as `@Transient` or `transient`.

@org.hibernate.annotations.AccessType

The annotation `@org.hibernate.annotations.AccessType` should be considered deprecated for FIELD and PROPERTY access. It is still useful however if you need to use a custom access type.

5.1.4.1.3. Optimistic lock

It is sometimes useful to avoid increasing the version number even if a given property is dirty (particularly collections). You can do that by annotating the property (or collection) with `@OptimisticLock(excluded=true)`.

More formally, specifies that updates to this property do not require acquisition of the optimistic lock.

5.1.4.1.4. Declaring column attributes

The column(s) used for a property mapping can be defined using the `@Column` annotation. Use it to override default values (see the JPA specification for more information on the defaults). You can use this annotation at the property level for properties that are:

- not annotated at all
- annotated with `@Basic`
- annotated with `@Version`
- annotated with `@Lob`
- annotated with `@Temporal`

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

The `name` property is mapped to the `flight_name` column, which is not nullable, has a length of 50 and is not updatable (making the property immutable).

This annotation can be applied to regular properties as well as `@Id` or `@Version` properties.

```
@Column(
    name="columnName";                                          (1)
    boolean unique() default false;                             (2)
    boolean nullable() default true;                            (3)
    boolean insertable() default true;                          (4)
    boolean updatable() default true;                           (5)
    String columnDefinition() default "";                       (6)
    String table() default "";                                  (7)
    int length() default 255;                                   (8)
    int precision() default 0; // decimal precision             (9)
    int scale() default 0; // decimal scale
```

(1) `name` (optional): the column name (default to the property name)
(2) `unique` (optional): set a unique constraint on this column or not (default false)
(3) `nullable` (optional): set the column as nullable (default true).
(4) `insertable` (optional): whether or not the column will be part of the insert statement (default true)

5. `updatable` (optional): whether or not the column will be part of the update statement (default true)
6. `columnDefinition` (optional): override the sql DDL fragment for this particular column (non portable)
7. `table` (optional): define the targeted table (default primary table)
8. `length` (optional): column length (default 255)
8. `precision` (optional): column decimal precision (default 0)
10. `scale` (optional): column decimal scale if useful (default 0)

5.1.4.1.5. Formula

Sometimes, you want the Database to do some computation for you rather than in the JVM, you might also create some kind of virtual column. You can use a SQL fragment (aka formula) instead of mapping a property into a column. This kind of property is read only (its value is calculated by your formula fragment).

```
@Formula("obj_length * obj_height * obj_width")
public long getObjectVolume()
```

The SQL fragment can be as complex as you want and even include subselects.

5.1.4.1.6. Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as @Basic
- Otherwise, if the type of the property is annotated as @Embeddable, it is mapped as @Embedded
- Otherwise, if the type of the property is `Serializable`, it is mapped as `@Basic` in a column holding the object in its serialized version
- Otherwise, if the type of the property is `java.sql.Clob` or `java.sql.Blob`, it is mapped as `@Lob` with the appropriate `LobType`

5.1.4.2. Property mapping with hbm.xml

`<property>` ############ JavaBean ##################

```
<property
        name="propertyName"                                    1
        column="column_name"                                   2
        type="typename"                                        3
        update="true|false"                                    4
        insert="true|false"                                    4
```

```
        formula="arbitrary SQL expression"                                  ❺
        access="field|property|ClassName"                                   ❻
        lazy="true|false"                                                   ❼
        unique="true|false"                                                 ❽
        not-null="true|false"                                               ❾
        optimistic-lock="true|false"                                        ❿
        generated="never|insert|always"                                     ⓫
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

❶ `name`# ###############

❷ `column`###### - ############### ################################ <column> ###### ######

❸ `type`######## Hibernate ########

❹ `update, insert` ###### - ###### `true` ## ############ SQL # `UPDATE` # `INSERT` #### ############### `false` ## ######################

❺ `formula`######## ## ################ SQL ##################################

❻ `access` (##### - ###### `property`): Hibernate ############################

❼ `lazy` (##### - ###### `false`): ## (## ######################)#

❽ `unique` (#####):############### DDL ############### `property-ref` ############### ###

❾ `not-null` (optional): enables the DDL generation of a nullability constraint for the columns.

❿ `optimistic-lock` (##### - ###### `true`): ## ##

⓫ `generated` (optional - defaults to `never`): specifies that this property value is actually generated by the database. See the discussion of *generated properties* for more information.

typename ############

1. Hibernate ######### `integer, string, character, date, timestamp, float, binary, serializable, object, blob` ##
2. ########## Java #### ## `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob` ##
3. ######### Java #######
4. ############ `com.illflow.type.MyCustomType` ##

########## Hibernate #### Hibernate ## Hibernate ####2, 3, 4######### getter ##

############ type ############## #### Hibernate.DATE # Hibernate.TIMESTAMP ###### ########################

access ######## Hibernate ################################## Hibernate ####### get/set ########### access="field" ######## Hibernate ########### get/set ############### ################## org.hibernate.property.PropertyAccessor ################## ############################

SQL ######## ###################### SQL #### SELECT ##############:

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

######(### customerId #########)## ###################### <formula> ########################

5.1.5. Embedded objects (aka components)

Embeddable objects (or components) are objects whose properties are mapped to the same table as the owning entity's table. Components can, in turn, declare their own properties, components or collections

It is possible to declare an embedded component inside an entity and even override its column mapping. Component classes have to be annotated at the class level with the `@Embeddable` annotation. It is possible to override the column mapping of an embedded object for a particular entity using the `@Embedded` and `@AttributeOverride` annotation in the associated property:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
            @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    } )
    Country bornIn;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
```

```
    String city;
    Country nationality; //no overriding here
}
```

```
@Embeddable
public class Country implements Serializable {
    private String iso2;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
    public void setIso2(String iso2) { this.iso2 = iso2; }


    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    ...
}
```

An embeddable object inherits the access type of its owning entity (note that you can override that using the `@Access` annotation).

The `Person` entity has two component properties, `homeAddress` and `bornIn`. `homeAddress` property has not been annotated, but Hibernate will guess that it is a persistent component by looking for the `@Embeddable` annotation in the Address class. We also override the mapping of a column name (to `bornCountryName`) with the `@Embedded` and `@AttributeOverride` annotations for each mapped attribute of `Country`. As you can see, `Country` is also a nested component of `Address`, again using auto-detection by Hibernate and JPA defaults. Overriding columns of embedded objects of embedded objects is through dotted expressions.

```
@Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="city", column = @Column(name="fld_city") ),
            @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
          @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
            //nationality columns in homeAddress are overridden
    } )
    Address homeAddress;
```

Hibernate Annotations supports something that is not explicitly supported by the JPA specification. You can annotate a embedded object with the `@MappedSuperclass` annotation to make the superclass properties persistent (see `@MappedSuperclass` for more informations).

You can also use association annotations in an embeddable object (ie `@OneToOne`, `@ManyToOne`, `@OneToMany` or `@ManyToMany`). To override the association columns you can use `@AssociationOverride`.

If you want to have the same embeddable object type twice in the same entity, the column name defaulting will not work as several embedded objects would share the same set of columns. In

plain JPA, you need to override at least one set of columns. Hibernate, however, allows you to enhance the default naming mechanism through the `NamingStrategy` interface. You can write a strategy that prevent name clashing in such a situation. `DefaultComponentSafeNamingStrategy` is an example of this.

If a property of the embedded object points back to the owning entity, annotate it with the `@Parent` annotation. Hibernate will make sure this property is properly loaded with the entity reference.

In XML, use the `<component>` element.

```
<component
        name="propertyName"                                    1
        class="className"                                      2
        insert="true|false"                                    3
        update="true|false"                                    4
        access="field|property|ClassName"                      5
        lazy="true|false"                                      6
        optimistic-lock="true|false"                           7
        unique="true|false"                                    8
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

1. `name`########
2. `class` ###### - ################################ ##################
3. `insert`############# SQL # `INSERT` ####################
4. `update`############# SQL # `UPDATE` ####################
5. `access` (##### - ###### `property`): Hibernate ############################
6. `lazy` (##### - ###### `false`): ## (########################)
7. `optimistic-lock` (##### - ###### `true`): ## ##
8. `unique` (##### - ###### `false`): ##

`<property>`

`<component>` ## `<parent>` ###########

The `<dynamic-component>` element allows a `Map` to be mapped as a component, where the property names refer to keys of the map. See ########### for more information. This feature is not supported in annotations.

5.1.6. Inheritance strategy

Java is a language supporting polymorphism: a class can inherit from another. Several strategies are possible to persist a class hierarchy:

- Single table per class hierarchy strategy: a single table hosts all the instances of a class hierarchy
- Joined subclass strategy: one table per class and subclass is present and each table persist the properties specific to a given subclass. The state of the entity is then stored in its corresponding class table and all its superclasses
- Table per class strategy: one table per concrete class and subclass is present and each table persist the properties of the class and its superclasses. The state of the entity is then stored entirely in the dedicated table for its class.

5.1.6.1. Single table per class hierarchy strategy

With this approach the properties of all the subclasses in a given mapped class hierarchy are stored in a single table.

Each subclass declares its own persistent properties and subclasses. Version and id properties are assumed to be inherited from the root class. Each subclass in a hierarchy must define a unique discriminator value. If this is not specified, the fully qualified Java class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, for the table-per-class-hierarchy mapping strategy, the `<subclass>` declaration is used. For example:

```
<subclass
        name="ClassName"                                    (1)
        discriminator-value="discriminator_value"           (2)
        proxy="ProxyInterface"                              (3)
        lazy="true|false"                                   (4)
        dynamic-update="true|false"
```

```
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

1. name###################
2. discriminator-value###### - ############ ###################
3. proxy (#####): ###################################
4. lazy (############ true): lazy="false" ####################

For information about inheritance mappings see *10########*.

5.1.6.1.1. discriminator

Discriminators are required for polymorphic persistence using the table-per-class-hierarchy mapping strategy. It declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Hibernate Core supports the follwoing restricted set of types as discriminator column: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

Use the `@DiscriminatorColumn` to define the discriminator column as well as the discriminator type.

##

`The enum DiscriminatorType used in javax.persitence.DiscriminatorColumn only contains the values STRING, CHAR and INTEGER which means that not all Hibernate supported types are available via the @DiscriminatorColumn annotation.`

You can also use `@DiscriminatorFormula` to express in SQL a virtual discriminator column. This is particularly useful when the discriminator value can be extracted from one or more columns of the table. Both `@DiscriminatorColumn` and `@DiscriminatorFormula` are to be set on the root entity (once per persisted hierarchy).

`@org.hibernate.annotations.DiscriminatorOptions` allows to optionally specify Hibernate specific discriminator options which are not standardized in JPA. The available options are `force` and `insert`. The `force` attribute is useful if the table contains rows with "extra" discriminator values that are not mapped to a persistent class. This could for example occur when working with a legacy database. If `force` is set to `true` Hibernate will specify the allowed discriminator values in the `SELECT` query, even when retrieving all instances of the root class. The second option - `insert` - tells Hibernate whether or not to include the discriminator column in SQL `INSERTs`. Usually the

column should be part of the `INSERT` statement, but if your discriminator column is also part of a mapped composite identifier you have to set this option to `false`.

####

There is also a `@org.hibernate.annotations.ForceDiscriminator` annotation which is deprecated since version 3.6. Use `@DiscriminatorOptions` instead.

Finally, use `@DiscriminatorValue` on each class of the hierarchy to specify the value stored in the discriminator column for a given entity. If you do not set `@DiscriminatorValue` on a class, the fully qualified class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, the `<discriminator>` element is used to define the discriminator column or formula:

```
<discriminator
        column="discriminator_column"                    (1)
        type="discriminator_type"                        (2)
        force="true|false"                               (3)
        insert="true|false"                              (4)
        formula="arbitrary sql expression"               (5)
/>
```

(1) `column`###### - ###### `class` ## #########

(2) `type` ###### - ###### `string` ## Hibernate ########

(3) `force` ###### - ###### `false` ## ################################ Hibernate ###### ##################

(4) `insert` ###### - ###### `true` ## ############################ `false` ########## (Hibernate # SQL # `INSERT` ###################)

(5) `formula` (#####) ################## SQL ####################

############ `<class>` # `<subclass>` ### `discriminator-value` ##########

`formula` ###################### SQL ########:

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

5.1.6.2. Joined subclass strategy

Each subclass can also be mapped to its own table. This is called the table-per-subclass mapping strategy. An inherited state is retrieved by joining with the table of the superclass. A discriminator column is not required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier. The primary key of this table is also a foreign key to the superclass table and described by the `@PrimaryKeyJoinColumn`s or the `<key>` element.

```
@Entity @Table(name="CATS")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cat implements Serializable {
    @Id @GeneratedValue(generator="cat-uuid")
    @GenericGenerator(name="cat-uuid", strategy="uuid")
    String getId() { return id; }

    ...
}

@Entity @Table(name="DOMESTIC_CATS")
@PrimaryKeyJoinColumn(name="CAT")
public class DomesticCat extends Cat {
    public String getName() { return name; }
}
```

##

The table name still defaults to the non qualified class name. Also if `@PrimaryKeyJoinColumn` is not set, the primary key / foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

In hbm.xml, use the `<joined-subclass>` element. For example:

```
<joined-subclass
        name="ClassName"                                    (1)
        table="tablename"                                   (2)
        proxy="ProxyInterface"                              (3)
        lazy="true|false"                                   (4)
        dynamic-update="true|false"
```

```
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

1. name####################
2. table :##############
3. proxy (#####): ###################################
4. lazy (############ true): lazy="false" ####################

Use the <key> element to declare the primary key / foreign key column. The mapping at the start of the chapter would then be re-written as:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>
```

```
</hibernate-mapping>
```

For information about inheritance mappings see *10########*.

5.1.6.3. Table per class strategy

A third option is to map only the concrete classes of an inheritance hierarchy to tables. This is called the table-per-concrete-class strategy. Each table defines all persistent states of the class, including the inherited state. In Hibernate, it is not necessary to explicitly map such inheritance hierarchies. You can map each class as a separate entity root. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable { ... }
```

Or in hbm.xml:

```
<union-subclass
        name="ClassName"                                    (1)
        table="tablename"                                   (2)
        proxy="ProxyInterface"                              (3)
        lazy="true|false"                                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

(1) name####################

(2) table :##############

(3) proxy (######): ####################################

(4) lazy (############ true): lazy="false" ####################

################################

For information about inheritance mappings see *10########*.

5.1.6.4. Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity). For that purpose you can map them as `@MappedSuperclass`.

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}

@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

In database, this hierarchy will be represented as an `Order` table having the `id`, `lastUpdate` and `lastUpdater` columns. The embedded superclass property mappings are copied into their entity subclasses. Remember that the embeddable superclass is not the root of the hierarchy though.

##

Properties from superclasses not mapped as `@MappedSuperclass` are ignored.

##

The default access type (field or methods) is used, unless you use the `@Access` annotation.

##

The same notion can be applied to `@Embeddable` objects to persist properties from their superclasses. You also need to use `@MappedSuperclass` to do that (this should not be considered as a standard EJB3 feature though)

##

It is allowed to mark a class as `@MappedSuperclass` in the middle of the mapped inheritance hierarchy.

##

Any class in the hierarchy non annotated with `@MappedSuperclass` nor `@Entity` will be ignored.

You can override columns defined in entity superclasses at the root entity level using the `@AttributeOverride` annotation.

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

    @Transient
    public int getMetricAltitude() {
        return metricAltitude;
    }

    @ManyToOne
    public PropulsionType getPropulsion() {
        return metricAltitude;
    }
    ...
}

@Entity
@AttributeOverride( name="altitude", column = @Column(name="fld_altitude") )
@AssociationOverride(
   name="propulsion",
   joinColumns = @JoinColumn(name="fld_propulsion_fk")
)
public class Plane extends FlyingObject {
    ...
}
```

The `altitude` property will be persisted in an `fld_altitude` column of table `Plane` and the propulsion association will be materialized in a `fld_propulsion_fk` foreign key column.

You can define `@AttributeOverride`(s) and `@AssociationOverride`(s) on `@Entity` classes, `@MappedSuperclass` classes and properties pointing to an `@Embeddable` object.

In hbm.xml, simply map the properties of the superclass in the `<class>` element of the entity that needs to inherit them.

5.1.6.5. Mapping one entity to several tables

While not recommended for a fresh schema, some legacy databases force your to map a single entity on several tables.

Using the `@SecondaryTable` or `@SecondaryTables` class level annotations. To express that a column is in a particular table, use the `table` parameter of `@Column` or `@JoinColumn`.

```
@Entity
@Table(name="MainCat")
@SecondaryTables({
    @SecondaryTable(name="Cat1", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")
    ),
    @SecondaryTable(name="Cat2", uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})
})
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In this example, `name` will be in `MainCat`. `storyPart1` will be in `Cat1` and `storyPart2` will be in `Cat2`. `Cat1` will be joined to `MainCat` using the `cat_id` as a foreign key, and `Cat2` using `id` (ie the same column name, the `MainCat` id column has). Plus a unique constraint on `storyPart2` has been set.

There is also additional tuning accessible via the `@org.hibernate.annotations.Table` annotation:

- `fetch`: If set to JOIN, the default, Hibernate will use an inner join to retrieve a secondary table defined by a class or its superclasses and an outer join for a secondary table defined by a subclass. If set to `SELECT` then Hibernate will use a sequential select for a secondary table defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a secondary defined by the class and its superclasses.
- `inverse`: If true, Hibernate will not try to insert or update the properties defined by this join. Default to false.
- `optional`: If enabled (the default), Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.
- `foreignKey`: defines the Foreign Key name of a secondary table pointing back to the primary table.

Make sure to use the secondary table name in the `appliesto` property

```
@Entity
@Table(name="MainCat")
@SecondaryTable(name="Cat1")
@org.hibernate.annotations.Table(
   appliesTo="Cat1",
   fetch=FetchMode.SELECT,
   optional=true)
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In hbm.xml, use the `<join>` element.

```
<join
        table="tablename"                                    1
        schema="owner"                                       2
        catalog="catalog"                                    3
        fetch="join|select"                                  4
        inverse="true|false"                                 5
        optional="true|false">                               6

        <key ... />

        <property ... />
        ...
</join>
```

1. `table` :###########
2. `schema` (#####): #### `<hibernate-mapping>` ########################
3. `catalog` (#####): #### `<hibernate-mapping>` ########################
4. `fetch` (##### - ###### `join`): `join` ######## Hibernate ########################## `<join>` ########################## `<join>` ################# `select` ########## Hibernate ############ `<join>` ## `<join>` ##############
5. `inverse` (##### - ###### `false`): ######### Hibernate ########################## #######
6. `optional` (##### - ###### `false`): ######### Hibernate ################# null ##### ###################################

###################################### (##############################):

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

###
#####################################

5.1.7. Mapping one to one and one to many associations

To link one entity to an other, you need to map the association property as a to one association. In the relational model, you can either use a foreign key or an association table, or (a bit less common) share the same primary key value between the two entities.

To mark an association, use either `@ManyToOne` or `@OnetoOne`.

`@ManyToOne` and `@OneToOne` have a parameter named `targetEntity` which describes the target entity name. You usually don't need this parameter since the default value (the type of the property that stores the association) is good in almost all cases. However this is useful when you want to use interfaces as the return type instead of the regular entity.

Setting a value of the `cascade` attribute to any meaningful value other than nothing will propagate certain operations to the associated object. The meaningful values are divided into three categories.

1. basic operations, which include: `persist, merge, delete, save-update, evict, replicate, lock and refresh`;
2. special values: `delete-orphan` or `all` ;
3. comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See ######### for a full explanation. Note that single valued many-to-one associations do not support orphan delete.

By default, single point associations are eagerly fetched in JPA 2. You can mark it as lazily fetched by using `@ManyToOne(fetch=FetchType.LAZY)` in which case Hibernate will proxy the association and load it when the state of the associated entity is reached. You can force Hibernate not to use a proxy by using `@LazyToOne(NO_PROXY)`. In this case, the property is fetched lazily when the instance variable is first accessed. This requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched.

With the default JPA options, single-ended associations are loaded with a subsequent select if set to `LAZY`, or a SQL JOIN is used for `EAGER` associations. You can however adjust the fetching strategy, ie how data is fetched by using `@Fetch`. `FetchMode` can be `SELECT` (a select is triggered when the association needs to be loaded) or `JOIN` (use a SQL JOIN to load the association while loading the owner entity). `JOIN` overrides any lazy attribute (an association loaded through a `JOIN` strategy cannot be lazy).

5.1.7.1. Using a foreign key or an association table

An ordinary association to another persistent class is declared using a

- `@ManyToOne` if several entities can point to the the target entity

- `@OneToOne` if only a single entity can point to the the target entity

and a foreign key in one table is referencing the primary key column(s) of the target table.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

The `@JoinColumn` attribute is optional, the default value(s) is the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column in the owned side. In this example `company_id` because the property name is `company` and the column id of Company is `id`.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}

public interface Company {
    ...
}
```

You can also map a to one association through an association table. This association table described by the `@JoinTable` annotation will contains a foreign key referencing back the entity table (through `@JoinTable.joinColumns`) and a a foreign key referencing the target entity table (through `@JoinTable.inverseJoinColumns`).

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumn(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
```

```
}
```

##

You can use a SQL fragment to simulate a physical join column using the `@JoinColumnOrFormula` / `@JoinColumnOrformulas` annotations (just like you can use a SQL fragment to simulate a property column via the `@Formula` annotation).

```
@Entity
public class Ticket implements Serializable {
    @ManyToOne
    @JoinColumnOrFormula(formula="(firstname + ' ' + lastname)")
    public Person getOwner() {
        return person;
    }
    ...
}
```

You can mark an association as mandatory by using the `optional=false` attribute. We recommend to use Bean Validation's `@NotNull` annotation as a better alternative however. As a consequence, the foreign key column(s) will be marked as not nullable (if possible).

When Hibernate cannot resolve the association because the expected associated element is not in database (wrong id on the association column), an exception is raised. This might be inconvenient for legacy and badly maintained schemas. You can ask Hibernate to ignore such elements instead of raising an exception using the `@NotFound` annotation.

#5.1 @NotFound annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    public Parent getParent() { ... }
    ...
}
```

Sometimes you want to delegate to your database the deletion of cascade when a given entity is deleted. In this case Hibernate generates a cascade delete constraint at the database level.

#5.2 @OnDelete annotation

```
@Entity
public class Child {
```

```
    ...
    @ManyToOne
    @OnDelete(action=OnDeleteAction.CASCADE)
    public Parent getParent() { ... }
    ...
}
```

Foreign key constraints, while generated by Hibernate, have a fairly unreadable name. You can override the constraint name using `@ForeignKey`.

#5.3 @ForeignKey annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @ForeignKey(name="FK_PARENT")
    public Parent getParent() { ... }
    ...
}

alter table Child add constraint FK_PARENT foreign key (parent_id) references Parent
```

Sometimes, you want to link one entity to an other not by the target entity primary key but by a different unique key. You can achieve that by referencing the unique key column(s) in `@JoinColumn.referenceColumnName`.

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
}
```

This is not encouraged however and should be reserved to legacy mappings.

In hbm.xml, mapping an association is similar. The main difference is that a `@OneToOne` is mapped as `<many-to-one unique="true"/>`, let's dive into the subject.

```
<many-to-one
        name="propertyName"                                         (1)
        column="column_name"                                        (2)
        class="ClassName"                                           (3)
        cascade="cascade_style"                                     (4)
        fetch="join|select"                                         (5)
        update="true|false"                                         (6)
        insert="true|false"                                         (6)
        property-ref="propertyNameFromAssociatedClass"              (7)
        access="field|property|ClassName"                           (8)
        unique="true|false"                                         (9)
        not-null="true|false"                                       (10)
        optimistic-lock="true|false"                                (11)
        lazy="proxy|no-proxy|false"                                 (12)
        not-found="ignore|exception"                                (13)
        entity-name="EntityName"                                    (14)
        formula="arbitrary SQL expression"                          (15)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

1. `name`########
2. `column` (#####):################# `<column>` ################
3. `class`###### - ################################# #########
4. `cascade`######## ###
5. `fetch`###### - ###### `select` ## ###################sequential select fetch######## #####
6. `update, insert`###### - ###### `true` ## ############ SQL # `UPDATE` ### `INSERT` ### ################ `false` ## ###############################
7. `property-ref`: (#####) ## ##
8. `access` (##### - ###### `property`): Hibernate ###########################
9. `unique`######## ##################### DDL ############## `property-ref` ###### ####################################
10. `not-null` (#####): ############ null ###### DDL ###########

(11) `optimistic-lock` (##### - ###### `true`): #####################################
##

(12) `lazy` (##### - ###### `proxy`): ########################## `lazy="no-proxy"` ## (####################
#)# `lazy="false"` #######################

(13) `not-found` ###### - ###### `exception`#: ##########################: `ignore` ####
#########################

(14) `entity-name` (#####):################

(15) `formula` (#####): ##### ############## SQL #

Setting a value of the `cascade` attribute to any meaningful value other than `none` will propagate certain operations to the associated object. The meaningful values are divided into three categories. First, basic operations, which include: `persist, merge, delete, save-update, evict, replicate, lock and refresh`; second, special values: `delete-orphan`; and third,`all` comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See ######### for a full explanation. Note that single valued, many-to-one and one-to-one, associations do not support orphan delete.

`many-to-one`

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

`property-ref` ##
######################## `Product` ##
`unique` ### SchemaExport ####### Hibernate # DDL ##########

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

`OrderItem`

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

##################

`<properties>`
########

###:

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

5.1.7.2. Sharing the primary key with the associated entity

The second approach is to ensure an entity and its associated entity share the same primary key. In this case the primary key column is also a foreign key and there is no extra column. These associations are always one to one.

#5.4 One to One association

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @MapsId
    public Heart getHeart() {
        return heart;
    }
    ...
}

@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

##

Many people got confused by these primary key based one to one associations. They can only be lazily loaded if Hibernate knows that the other side of the association is always present. To indicate to Hibernate that it is the case, use `@OneToOne(optional=false)`.

In hbm.xml, use the following mapping.

```
<one-to-one
        name="propertyName"                                          1
        class="ClassName"                                            2
        cascade="cascade_style"                                      3
        constrained="true|false"                                     4
        fetch="join|select"                                          5
        property-ref="propertyNameFromAssociatedClass"               6
        access="field|property|ClassName"                            7
        formula="any SQL expression"                                 8
```

```
        lazy="proxy|no-proxy|false"                                    (9)
        entity-name="EntityName"                                       (10)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

1. name########
2. class###### - ############################## #########
3. cascade######## ##
4. constrained######## ## ####### save() # delete() ### ######## ###################
5. fetch###### - ###### select ## ##################sequential select fetch######## #####
6. property-ref######## ## ##
7. access (##### - ###### property): Hibernate ############################
8. formula (#####): ## ######### SQL ########################## org.hibernate.test.onetooneformula ###########
9. lazy (##### - ###### proxy): ########################## lazy="no-proxy" ## (################### #)# lazy="false" ###################### ## *constrained="false"* ########### ##############################
10. entity-name (#####):################

##############################2################2#########################2## ##

####################### Employee # Person ###############

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

PERSON # EMPLOYEE ## foreign ## #### Hibernate

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
```

```
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

Employee ######## Person # employee ######################### Person ############ ###################### Person ########## Person # employee ########## Employee #### ####################

5.1.8. natural-id

Although we recommend the use of surrogate keys as primary keys, you should try to identify natural keys for all entities. A natural key is a property or combination of properties that is unique and non-null. It is also immutable. Map the properties of the natural key as `@NaturalId` or map them inside the `<natural-id>` element. Hibernate will generate the necessary unique key and nullability constraints and, as a result, your mapping will be more self-documenting.

```
@Entity
public class Citizen {
    @Id
    @GeneratedValue
    private Integer id;
    private String firstname;
    private String lastname;

    @NaturalId
    @ManyToOne
    private State state;

    @NaturalId
    private String ssn;
    ...
}



//and later on query
List results = s.createCriteria( Citizen.class )
                .add( Restrictions.naturalId().set( "ssn", "1234" ).set( "state", ste ) )
                .list();
```

Or in XML,

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
```

```
</natural-id>
```

###################### `equals()` # `hashCode()` ##############

######################################

- `mutable` (###### ###### `false`): ######################(##)###########

5.1.9. Any

There is one more type of property mapping. The `@Any` mapping defines a polymorphic association to classes from multiple tables. This type of mapping requires more than one column. The first column contains the type of the associated entity. The remaining columns contain the identifier. It is impossible to specify a foreign key constraint for this kind of association. This is not the usual way of mapping polymorphic associations and you should use this only in special cases. For example, for audit logs, user session data, etc.

The `@Any` annotation describes the column holding the metadata information. To link the value of the metadata information and an actual entity type, The `@AnyDef` and `@AnyDefs` annotations are used. The `metaType` attribute allows the application to specify a custom type that maps database column values to persistent classes that have identifier properties of the type specified by `idType`. You must specify the mapping from values of the `metaType` to class names.

```
@Any( metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
@AnyMetaDef(
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
@JoinColumn( name = "property_id" )
public Property getMainProperty() {
    return mainProperty;
}
```

Note that `@AnyDef` can be mutualized and reused. It is recommended to place it as a package metadata in this case.

```
//on a package
@AnyMetaDef( name="property"
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
package org.hibernate.test.annotations.any;
```

```
//in a class
    @Any( metaDef="property", metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
    @JoinColumn( name = "property_id" )
    public Property getMainProperty() {
        return mainProperty;
    }
```

The hbm.xml equivalent is:

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

##

You cannot mutualize the metadata in hbm.xml as you can in annotations.

```
<any
        name="propertyName"                     (1)
        id-type="idtypename"                    (2)
        meta-type="metatypename"                (3)
        cascade="cascade_style"                 (4)
        access="field|property|ClassName"       (5)
        optimistic-lock="true|false"            (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

1. name# #######
2. id-type# ######
3. meta-type###### - ###### string ## ######################
4. cascade###### - ###### none ## ############
5. access (##### - ###### property): Hibernate ############################

6. optimistic-lock (##### - ###### true): ## ##

5.1.10.

<properties> ## property-ref ###

```
<properties
        name="logicalName"                                          1
        insert="true|false"                                         2
        update="true|false"                                         3
        optimistic-lock="true|false"                                4
        unique="true|false"                                         5
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

1. name : ######################## ##### #
2. insert############## SQL # INSERT ####################
3. update############## SQL # UPDATE ####################
4. optimistic-lock (##### - ###### true): ## ##
5. unique (##### - ###### false): ##

############ <properties> ############

```
<class name="Person">
    <id name="personNumber"/>

    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

######### Person ##

```
<many-to-one name="owner"
        class="Person" property-ref="name">
```

```
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

##

When using annotations as a mapping strategy, such construct is not necessary as the binding between a column and its related column on the associated table is done directly

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
}
```

##

5.1.11. Some hbm.xml specificities

The hbm.xml structure has some specificities naturally not present when using annotations, let's describe them briefly.

5.1.11.1. Doctype

XML ###################################### DTD ##### URL # `hibernate-x.x.x/src/org/hibernate` ########## `hibernate.jar` ####### Hibernate ############## DTD ###### ############ DTD ########################## DTD ############

5.1.11.1.1.

######## Hibernate ########## DTD ########### `org.xml.sax.EntityResolver` ######## XML ############ SAXReader ############ DTD ############## `EntityResolver` #2## ####### ID ###########

- a `hibernate namespace` is recognized whenever the resolver encounters a systemId starting with `http://www.hibernate.org/dtd/`. The resolver attempts to resolve these entities via the classloader which loaded the Hibernate classes.
- `user namespace` ####### URL ###### `classpath://` ######## ID #################### ##### (1) ########################### (2) Hibernate ############################### ##############

#####################

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

Where `types.xml` is a resource in the `your.domain` package and contains a custom *typedef*.

5.1.11.2. Hibernate-mapping

######################## `schema` ### `catalog` ##################################### ####(###)### ######################## `default-cascade` #### `cascade` ########################## ######################## `auto-import` ###

```
<hibernate-mapping
         schema="schemaName"                              (1)
         catalog="catalogName"                            (2)
         default-cascade="cascade_style"                  (3)
         default-access="field|property|ClassName"        (4)
         default-lazy="true|false"                        (5)
         auto-import="true|false"                         (6)
         package="package.name"                           (7)
 />
```

(1) `schema`######################

2. `catalog` ######################
3. `default-cascade` ###### - ###### none## ################
4. `default-access` (##### - ###### `property` ## Hibernate ###################### `PropertyAccessor` #################
5. `default-lazy` (##### - ###### `true`)# `lazy` ##################################### ##
6. `auto-import` ###### - ###### `true`## #########
7. `package` (#####): ## (prefix) #######

####################2###### `auto-import="false"` #########2######"########"#### ############ Hibernate ##########

`hibernate-mapping` ###################### `<class>` ############################## ############(############)##(########## ####################)################ `Cat.hbm.xml` , `Dog.hbm.xml` , ########## `Animal.hbm.xml` #

5.1.11.3. Key

The `<key>` element is featured a few times within this guide. It appears anywhere the parent mapping element defines a join to a new table that references the primary key of the original table. It also defines the foreign key in the joined table:

```
<key
        column="columnname"                                    1
        on-delete="noaction|cascade"                           2
        property-ref="propertyName"                            3
        not-null="true|false"                                  4
        update="true|false"                                    5
        unique="true|false"                                    6
/>
```

1. `column` (#####):################ `<column>` ###############
2. `on-delete` (#####, ###### `noaction`): ######################################
3. `property-ref` (#####): ################################### (################ #)#
4. `not-null` (#####): ######## null ################ (############################## #)#
5. `update` (#####): ########################## (##############################)#
6. `unique` (#####): ############################ (############################## #)#

################################ on-delete="cascade" ###################### Hibernate ## DELETE ######################## ON CASCADE DELETE ################### ############ Hibernate ##################################

not-null # update ################################ null ###################### <key not-null="true"> ############ ########## #

5.1.11.4. Import

##############2########### Hibernate ### ### auto-import="true" ## ##################

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"                    (1)
        rename="ShortName"                   (2)
/>
```

(1) class# Java #################

(2) rename ###### - ##################### ##############

##

This feature is unique to hbm.xml and is not supported in annotations.

5.1.11.5. column # formula

column ###################### <column> ############### <formula> # formula ##### #####

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"
```

```
        read="SQL expression"
        write="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

Most of the attributes on `column` provide a means of tailoring the DDL during automatic schema generation. The `read` and `write` attributes allow you to specify custom SQL that Hibernate will use to access the column's value. For more on this, see the discussion of *column read and write expressions*.

The `column` and `formula` elements can even be combined within the same property or association mapping to express, for example, exotic join conditions.

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

5.2. Hibernate

5.2.1.

In relation to the persistence service, Java language-level objects are classified into two groups:

Java ## ###### ####################### ### ODMG ################################ ###

######################## # ## (############# #)### ##### ## ### ###

`java.lang.String` ################################### JDK ############# Java ## (### #) ## ###

##############################

Java ###### (######################) # SQL /############################## Hibernate ############################## <class> # <subclass> ################ <property> # <component> ###### type ############### Hibernate # ###### ###### Hibernate # (## JDK #######) ###

############## Hibernate ####### null ################

5.2.2.

##################

integer, long, short, float, double, character, byte, boolean, yes_no, true_false
: Java ############################## SQL ############# boolean, yes_no # true_false ##### Java # boolean ### java.lang.Boolean ###########

string
: java.lang.String ## VARCHAR #### Oracle # VARCHAR2 ##########

date, time, timestamp
: java.util.Date ########## SQL ## DATE # TIME # TIMESTAMP ############## ##### ####

calendar, calendar_date
: java.util.Calendar ## SQL # ## TIMESTAMP # DATE (######################

big_decimal, big_integer
: java.math.BigDecimal # java.math.BigInteger ## NUMERIC#### Oracle # NUMBER #### ######

locale, timezone, currency
: java.util.Locale # java.util.TimeZone # java.util.Currency ## VARCHAR #### Oracle # VARCHAR2 ########## Locale # Currency ############# ISO ############## TimeZone ############# ID ###########

class
: java.lang.Class ## VARCHAR #### Oracle # VARCHAR2 ########## Class ########### ###########

binary
: ########## SQL #################

text
: Maps long Java strings to a SQL LONGVARCHAR or TEXT type.

image
: Maps long byte arrays to a SQL LONGVARBINARY.

`serializable`
: ######### Java ###### SQL ## Java ##### ################### Hibernate ### `serializable` ###########

`clob, blob`
: JDBC ### `java.sql.Clob` # `java.sql.Blob` ########### blob # clob ################ ##

`materialized_clob`
: Maps long Java strings to a SQL `CLOB` type. When read, the `CLOB` value is immediately materialized into a Java string. Some drivers require the `CLOB` value to be read within a transaction. Once materialized, the Java string is available outside of the transaction.

`materialized_blob`
: Maps long Java byte arrays to a SQL `BLOB` type. When read, the `BLOB` value is immediately materialized into a byte array. Some drivers require the `BLOB` value to be read within a transaction. Once materialized, the byte array is available outside of the transaction.

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary`
: ############# Java ############# Hibernate #### Java ######################## ###################### `imm_timestamp` ################### `Date.setTime()` ##### ## (#####) ########### ##########

######################## `binary` # `blob` # `clob` ################################### #########

######### `org.hibernate.Hibernate` ###### `Type` ################ `Hibernate.STRING` # `string` ##########

5.2.3.

############################## `java.lang.BigInteger` ######## `VARCHAR` ########## ######## Hibernate ##1######## ############################## `java.lang.String` ## `getName()` / `setName()` Java ###### `FIRST_NAME` # `INITIAL` # `SURNAME` ############

############ `org.hibernate.UserType` ### `org.hibernate.CompositeUserType` ####### ## `org.hibernate.test.DoubleStringType` ##########

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

<column> ##

CompositeUserType # EnhancedUserType # UserCollectionType # UserVersionType ####### #############################

############ UserType ####################### UserType # org.hibernate.usertype.ParameterizedType ## ####### <type> ##########

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

UserType ########### Properties ######### default ##############################

UserType ################################### <typedef> #################### Typedefs

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

########################### typedef ####################################

Even though Hibernate's rich range of built-in types and support for components means you will rarely need to use a custom type, it is considered good practice to use custom types for non-entity classes that occur frequently in your application. For example, a `MonetaryAmount` class is a good candidate for a `CompositeUserType`, even though it could be mapped as a component. One reason for this is abstraction. With a custom type, your mapping documents would be protected against changes to the way monetary values are represented.

5.3.

(##########################)# Hibernate ###########################

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
```

```
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

class ##### entity-name

> **##**
>
> This feature is not supported in Annotations

5.4. ########### SQL

Hibernate ###### SQL #################### Hibernate # SQL # Dialect ###################################### SQL Server ####### MySQL

```
@Entity @Table(name="`Line Item`")
class LineItem {
   @id @Column(name="`Item Id`") Integer id;
   @Column(name="`Item #`") int itemNumber
}

<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5.

Hibernate ############################# ########## ###### ## # Hibernate ## Hibernate # INSERT # UPDATE # SQL ######################### SELECT SQL

Properties marked as generated must additionally be non-insertable and non-updateable. Only *versions*, *timestamps*, and *simple properties*, can be marked as generated.

`never` (#####) - ##

`insert`: the given property value is generated on insert, but is not regenerated on subsequent updates. Properties like created-date fall into this category. Even though *version* and *timestamp* properties can be marked as generated, this option is not available.

`always` - ##############################

To mark a property as generated, use `@Generated`.

5.6. Column transformers: read and write expressions

Hibernate allows you to customize the SQL it uses to read and write the values of columns mapped to *simple properties*. For example, if your database provides a set of data encryption functions, you can invoke them for individual columns like this:

```
@Entity
class CreditCard {
   @Column(name="credit_card_num")
   @ColumnTransformer(
      read="decrypt(credit_card_num)",
      write="encrypt(?)")
   public String getCreditCardNumber() { return creditCardNumber; }
   public void setCreditCardNumber(String number) { this.creditCardNumber = number; }
   private String creditCardNumber;
}
```

or in XML

```
<property name="creditCardNumber">
        <column
          name="credit_card_num"
          read="decrypt(credit_card_num)"
          write="encrypt(?)"/>
</property>
```

##

You can use the plural form `@ColumnTransformers` if more than one columns need to define either of these rules.

If a property uses more that one column, you must use the `forColumn` attribute to specify which column, the expressions are targeting.

```
@Entity
class User {
```

```
    @Type(type="com.acme.type.CreditCardType")
    @Columns( {
       @Column(name="credit_card_num"),
       @Column(name="exp_date") } )
    @ColumnTransformer(
       forColumn="credit_card_num",
       read="decrypt(credit_card_num)",
       write="encrypt(?)")
    public CreditCard getCreditCard() { return creditCard; }
    public void setCreditCard(CreditCard card) { this.creditCard = card; }
    private CreditCard creditCard;
}
```

Hibernate applies the custom expressions automatically whenever the property is referenced in a query. This functionality is similar to a derived-property `formula` with two differences:

- The property is backed by one or more columns that are exported as part of automatic schema generation.
- The property is read-write, not read-only.

The `write` expression, if specified, must contain exactly one '?' placeholder for the value.

5.7.

Hibernate ## CREATE # DROP #### Hibernate ### #################################### `java.sql.Statement.execute()` ############ #### SQL #####ALTER#INSERT####################################2############ ##

1####### CREATE # DROP ##############################:

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

2####### CREATE # DROP ## `org.hibernate.mapping.AuxiliaryDatabaseObject` #####################

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
```

```
</hibernate-mapping>
```

##

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9iDialect"/>
        <dialect-scope name="org.hibernate.dialect.Oracle10gDialect"/>
    </database-object>
</hibernate-mapping>
```

##

This feature is not supported in Annotations

Types

As an Object/Relational Mapping solution, Hibernate deals with both the Java and JDBC representations of application data. An online catalog application, for example, most likely has `Product` object with a number of attributes such as a `sku`, `name`, etc. For these individual attributes, Hibernate must be able to read the values out of the database and write them back. This 'marshalling' is the function of a *Hibernate type*, which is an implementation of the `org.hibernate.type.Type` interface. In addition, a *Hibernate type* describes various aspects of behavior of the Java type such as "how is equality checked?" or "how are values cloned?".

####

A Hibernate type is neither a Java type nor a SQL datatype; it provides a information about both.

When you encounter the term *type* in regards to Hibernate be aware that usage might refer to the Java type, the SQL/JDBC type or the Hibernate type.

Hibernate categorizes types into two high-level groups: value types (see *#Value types#*) and entity types (see *#Entity types#*).

6.1. Value types

The main distinguishing characteristic of a value type is the fact that they do not define their own lifecycle. We say that they are "owned" by something else (specifically an entity, as we will see later) which defines their lifecycle. Value types are further classified into 3 sub-categories: basic types (see *#Basic value types#*), composite types (see *#Composite types#*) amd collection types (see *#Collection types#*).

6.1.1. Basic value types

The norm for basic value types is that they map a single database value (column) to a single, non-aggregated Java type. Hibernate provides a number of built-in basic types, which we will present in the following sections by the Java type. Mainly these follow the natural mappings recommended in the JDBC specification. We will later cover how to override these mapping and how to provide and use alternative type mappings.

6.1.1.1. java.lang.String

`org.hibernate.type.StringType`
: Maps a string to the JDBC VARCHAR type. This is the standard mapping for a string if no Hibernate type is specified.

  Registered under `string` and `java.lang.String` in the type registry (see *#Type registry#*).

`org.hibernate.type.MaterializedClob`
: Maps a string to a JDBC CLOB type

  Registered under `materialized_clob` in the type registry (see *#Type registry#*).

`org.hibernate.type.TextType`
: Maps a string to a JDBC LONGVARCHAR type

  Registered under `text` in the type registry (see *#Type registry#*).

6.1.1.2. `java.lang.Character` (or char primitive)

`org.hibernate.type.CharacterType`
: Maps a char or `java.lang.Character` to a JDBC CHAR

  Registered under `char` and `java.lang.Character` in the type registry (see *#Type registry#*).

6.1.1.3. `java.lang.Boolean` (or boolean primitive)

`org.hibernate.type.BooleanType`
: Maps a boolean to a JDBC BIT type

  Registered under `boolean` and `java.lang.Boolean` in the type registry (see *#Type registry#*).

`org.hibernate.type.NumericBooleanType`
: Maps a boolean to a JDBC INTEGER type as 0 = false, 1 = true

  Registered under `numeric_boolean` in the type registry (see *#Type registry#*).

`org.hibernate.type.YesNoType`
: Maps a boolean to a JDBC CHAR type as ('N' | 'n') = false, ('Y' | 'y') = true

  Registered under `yes_no` in the type registry (see *#Type registry#*).

`org.hibernate.type.TrueFalseType`
: Maps a boolean to a JDBC CHAR type as ('F' | 'f') = false, ('T' | 't') = true

  Registered under `true_false` in the type registry (see *#Type registry#*).

6.1.1.4. `java.lang.Byte` (or byte primitive)

`org.hibernate.type.ByteType`
: Maps a byte or `java.lang.Byte` to a JDBC TINYINT

  Registered under `byte` and `java.lang.Byte` in the type registry (see *#Type registry#*).

6.1.1.5. `java.lang.Short` (or short primitive)

`org.hibernate.type.ShortType`
: Maps a short or `java.lang.Short` to a JDBC SMALLINT

Registered under `short` and `java.lang.Short` in the type registry (see *#Type registry#*).

6.1.1.6. `java.lang.Integer` (or int primitive)

`org.hibernate.type.IntegerTypes`
Maps an int or `java.lang.Integer` to a JDBC INTEGER

Registered under `int` and `java.lang.Integer`in the type registry (see *#Type registry#*).

6.1.1.7. `java.lang.Long` (or long primitive)

`org.hibernate.type.LongType`
Maps a long or `java.lang.Long` to a JDBC BIGINT

Registered under `long` and `java.lang.Long` in the type registry (see *#Type registry#*).

6.1.1.8. `java.lang.Float` (or float primitive)

`org.hibernate.type.FloatType`
Maps a float or `java.lang.Float` to a JDBC FLOAT

Registered under `float` and `java.lang.Float` in the type registry (see *#Type registry#*).

6.1.1.9. `java.lang.Double` (or double primitive)

`org.hibernate.type.DoubleType`
Maps a double or `java.lang.Double` to a JDBC DOUBLE

Registered under `double` and `java.lang.Double` in the type registry (see *#Type registry#*).

6.1.1.10. `java.math.BigInteger`

`org.hibernate.type.BigIntegerType`
Maps a `java.math.BigInteger` to a JDBC NUMERIC

Registered under `big_integer` and `java.math.BigInteger` in the type registry (see *#Type registry#*).

6.1.1.11. `java.math.BigDecimal`

`org.hibernate.type.BigDecimalType`
Maps a `java.math.BigDecimal` to a JDBC NUMERIC

Registered under `big_decimal` and `java.math.BigDecimal` in the type registry (see *#Type registry#*).

6.1.1.12. `java.util.Date` or `java.sql.Timestamp`

`org.hibernate.type.TimestampType`
: Maps a `java.sql.Timestamp` to a JDBC TIMESTAMP

  Registered under `timestamp`, `java.sql.Timestamp` and `java.util.Date` in the type registry (see *#Type registry#*).

6.1.1.13. `java.sql.Time`

`org.hibernate.type.TimeType`
: Maps a `java.sql.Time` to a JDBC TIME

  Registered under `time` and `java.sql.Time` in the type registry (see *#Type registry#*).

6.1.1.14. `java.sql.Date`

`org.hibernate.type.DateType`
: Maps a `java.sql.Date` to a JDBC DATE

  Registered under `date` and `java.sql.Date` in the type registry (see *#Type registry#*).

6.1.1.15. `java.util.Calendar`

`org.hibernate.type.CalendarType`
: Maps a `java.util.Calendar` to a JDBC TIMESTAMP

  Registered under `calendar`, `java.util.Calendar` and `java.util.GregorianCalendar` in the type registry (see *#Type registry#*).

`org.hibernate.type.CalendarDateType`
: Maps a `java.util.Calendar` to a JDBC DATE

  Registered under `calendar_date` in the type registry (see *#Type registry#*).

6.1.1.16. `java.util.Currency`

`org.hibernate.type.CurrencyType`
: Maps a `java.util.Currency` to a JDBC VARCHAR (using the Currency code)

  Registered under `currency` and `java.util.Currency` in the type registry (see *#Type registry#*).

6.1.1.17. `java.util.Locale`

`org.hibernate.type.LocaleType`
: Maps a `java.util.Locale` to a JDBC VARCHAR (using the Locale code)

Registered under `locale` and `java.util.Locale` in the type registry (see *#Type registry#*).

6.1.1.18. `java.util.TimeZone`

`org.hibernate.type.TimeZoneType`
Maps a `java.util.TimeZone` to a JDBC VARCHAR (using the TimeZone ID)

Registered under `timezone` and `java.util.TimeZone` in the type registry (see *#Type registry#*).

6.1.1.19. `java.net.URL`

`org.hibernate.type.UrlType`
Maps a `java.net.URL` to a JDBC VARCHAR (using the external form)

Registered under `url` and `java.net.URL` in the type registry (see *#Type registry#*).

6.1.1.20. `java.lang.Class`

`org.hibernate.type.ClassType`
Maps a `java.lang.Class` to a JDBC VARCHAR (using the Class name)

Registered under `class` and `java.lang.Class` in the type registry (see *#Type registry#*).

6.1.1.21. `java.sql.Blob`

`org.hibernate.type.BlobType`
Maps a `java.sql.Blob` to a JDBC BLOB

Registered under `blob` and `java.sql.Blob` in the type registry (see *#Type registry#*).

6.1.1.22. `java.sql.Clob`

`org.hibernate.type.ClobType`
Maps a `java.sql.Clob` to a JDBC CLOB

Registered under `clob` and `java.sql.Clob` in the type registry (see *#Type registry#*).

6.1.1.23. byte[]

`org.hibernate.type.BinaryType`
Maps a primitive byte[] to a JDBC VARBINARY

Registered under `binary` and `byte[]` in the type registry (see *#Type registry#*).

`org.hibernate.type.MaterializedBlobType`
Maps a primitive byte[] to a JDBC BLOB

Registered under `materialized_blob` in the type registry (see *#Type registry#*).

`org.hibernate.type.ImageType`
Maps a primitive byte[] to a JDBC LONGVARBINARY

Registered under `image` in the type registry (see *#Type registry#*).

6.1.1.24. Byte[]

`org.hibernate.type.BinaryType`
Maps a java.lang.Byte[] to a JDBC VARBINARY

Registered under `wrapper-binary`, `Byte[]` and `java.lang.Byte[]` in the type registry (see *#Type registry#*).

6.1.1.25. char[]

`org.hibernate.type.CharArrayType`
Maps a char[] to a JDBC VARCHAR

Registered under `characters` and `char[]` in the type registry (see *#Type registry#*).

6.1.1.26. Character[]

`org.hibernate.type.CharacterArrayType`
Maps a java.lang.Character[] to a JDBC VARCHAR

Registered under `wrapper-characters`, `Character[]` and `java.lang.Character[]` in the type registry (see *#Type registry#*).

6.1.1.27. `java.util.UUID`

`org.hibernate.type.UUIDBinaryType`
Maps a java.util.UUID to a JDBC BINARY

Registered under `uuid-binary` and `java.util.UUID` in the type registry (see *#Type registry#*).

`org.hibernate.type.UUIDCharType`
Maps a java.util.UUID to a JDBC CHAR (though VARCHAR is fine too for existing schemas)

Registered under `uuid-char` in the type registry (see *#Type registry#*).

`org.hibernate.type.PostgresUUIDType`
: Maps a java.util.UUID to the PostgreSQL UUID data type (through `Types#OTHER` which is how the PostgreSQL JDBC driver defines it).

  Registered under `pg-uuid` in the type registry (see *#Type registry#*).

6.1.1.28. `java.io.Serializable`

`org.hibernate.type.SerializableType`
: Maps implementors of java.lang.Serializable to a JDBC VARBINARY

  Unlike the other value types, there are multiple instances of this type. It gets registered once under `java.io.Serializable`. Additionally it gets registered under the specific `java.io.Serializable` implementation class names.

6.1.2. Composite types

##

The Java Persistence API calls these embedded types, while Hibernate traditionally called them components. Just be aware that both terms are used and mean the same thing in the scope of discussing Hibernate.

Components represent aggregations of values into a single Java type. For example, you might have an Address class that aggregates street, city, state, etc information or a Name class that aggregates the parts of a person's Name. In many ways a component looks exactly like an entity. They are both (generally speaking) classes written specifically for the application. They both might have references to other application-specific classes, as well as to collections and simple JDK types. As discussed before, the only distinguishing factory is the fact that a component does not own its own lifecycle nor does it define an identifier.

6.1.3. Collection types

####

It is critical understand that we mean the collection itself, not its contents. The contents of the collection can in turn be basic, component or entity types (though not collections), but the collection itself is owned.

Collections are covered in *7##############*.

6.2. Entity types

The definition of entities is covered in detail in *4######*. For the purpose of this discussion, it is enough to say that entities are (generally application-specific) classes which correlate to rows

in a table. Specifically they correlate to the row by means of a unique identifier. Because of this unique identifier, entities exist independently and define their own lifecycle. As an example, when we delete a `Membership`, both the `User` and `Group` entities remain.

##

This notion of entity independence can be modified by the application developer using the concept of cascades. Cascades allow certain operations to continue (or "cascade") across an association from one entity to another. Cascades are covered in detail in 8#########.

6.3. Significance of type categories

Why do we spend so much time categorizing the various types of types? What is the significance of the distinction?

The main categorization was between entity types and value types. To review we said that entities, by nature of their unique identifier, exist independently of other objects whereas values do not. An application cannot "delete" a Product sku; instead, the sku is removed when the Product itself is deleted (obviously you can *update* the sku of that Product to null to make it "go away", but even there the access is done through the Product).

Nor can you define an association *to* that Product sku. You *can* define an association to Product *based on* its sku, assuming sku is unique, but that is totally different.

TBC...

6.4. Custom types

Hibernate makes it relatively easy for developers to create their own *value* types. For example, you might want to persist properties of type `java.lang.BigInteger` to `VARCHAR` columns. Custom types are not limited to mapping values to a single table column. So, for example, you might want to concatenate together `FIRST_NAME`, `INITIAL` and `SURNAME` columns into a `java.lang.String`.

There are 3 approaches to developing a custom Hibernate type. As a means of illustrating the different approaches, lets consider a use case where we need to compose a `java.math.BigDecimal` and `java.util.Currency` together into a custom `Money` class.

6.4.1. Custom types using `org.hibernate.type.Type`

The first approach is to directly implement the `org.hibernate.type.Type` interface (or one of its derivatives). Probably, you will be more interested in the more specific `org.hibernate.type.BasicType` contract which would allow registration of the type (see *#Type registry#*). The benefit of this registration is that whenever the metadata for a particular property does not specify the Hibernate type to use, Hibernate will consult the registry for the exposed

property type. In our example, the property type would be `Money`, which is the key we would use to register our type in the registry:

#6.1 Defining and registering the custom Type

```
public class MoneyType implements BasicType {
    public String[] getRegistrationKeys() {
        return new String[] { Money.class.getName() };
    }

        public int[] sqlTypes(Mapping mapping) {
             // We will simply use delegation to the standard basic types for BigDecimal and
 Currency for many of the
            // Type methods...
            return new int[] {
                    BigDecimalType.INSTANCE.sqlType(),
                    CurrencyType.INSTANCE.sqlType(),
            };
            // we could also have honored any registry overrides via...
            //return new int[] {
                                                    //
 mappings.getTypeResolver().basic( BigDecimal.class.getName() ).sqlTypes( mappings )[0],
                                                    //
 mappings.getTypeResolver().basic( Currency.class.getName() ).sqlTypes( mappings )[0]
            //};
        }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLEx
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, boolean[] settable, SessionImplementor
            throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}

Configuration cfg = new Configuration();
```

```
cfg.registerTypeOverride( new MoneyType() );
cfg...;
```

####

It is important that we registered the type *before* adding mappings.

6.4.2. Custom types using `org.hibernate.usertype.UserType`

##

Both `org.hibernate.usertype.UserType` and `org.hibernate.usertype.CompositeUserType` were originally added to isolate user code from internal changes to the `org.hibernate.type.Type` interfaces.

The second approach is the use the `org.hibernate.usertype.UserType` interface, which presents a somewhat simplified view of the `org.hibernate.type.Type` interface. Using a `org.hibernate.usertype.UserType`, our `Money` custom type would look as follows:

#6.2 Defining the custom UserType

```
public class MoneyType implements UserType {
    public int[] sqlTypes() {
        return new int[] {
                BigDecimalType.INSTANCE.sqlType(),
                CurrencyType.INSTANCE.sqlType(),
        };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, Object owner) throws SQLException {
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

   public void nullSafeSet(PreparedStatement st, Object value, int index) throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
```

```
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

There is not much difference between the `org.hibernate.type.Type` example and the `org.hibernate.usertype.UserType` example, but that is only because of the snippets shown. If you choose the `org.hibernate.type.Type` approach there are quite a few more methods you would need to implement as compared to the `org.hibernate.usertype.UserType`.

6.4.3. Custom types using `org.hibernate.usertype.CompositeUserType`

The third and final approach is the use the `org.hibernate.usertype.CompositeUserType` interface, which differs from `org.hibernate.usertype.UserType` in that it gives us the ability to provide Hibernate the information to handle the composition within the `Money` class (specifically the 2 attributes). This would give us the capability, for example, to reference the `amount` attribute in an HQL query. Using a `org.hibernate.usertype.CompositeUserType`, our `Money` custom type would look as follows:

#6.3 Defining the custom CompositeUserType

```
public class MoneyType implements CompositeUserType {
    public String[] getPropertyNames() {
          // ORDER IS IMPORTANT!   it must match the order the columns are defined in the
 property mapping
        return new String[] { "amount", "currency" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { BigDecimalType.INSTANCE, CurrencyType.INSTANCE };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object getPropertyValue(Object component, int propertyIndex) {
        if ( component == null ) {
            return null;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                return money.getAmount();
            }
            case 1: {
                return money.getCurrency();
            }
            default: {
```

```
                throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
    }

        public void setPropertyValue(Object component, int propertyIndex, Object value) throws HibernateException
        if ( component == null ) {
            return;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                money.setAmount( (BigDecimal) value );
                break;
            }
            case 1: {
                money.setCurrency( (Currency) value );
                break;
            }
            default: {
               throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
        }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLEx
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor session) throws SQLE
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

6.5. Type registry

Internally Hibernate uses a registry of basic types (see *#Basic value types#*) when it needs to resolve the specific `org.hibernate.type.Type` to use in certain situations. It also provides a way for applications to add extra basic type registrations as well as override the standard basic type registrations.

To register a new type or to override an existing type registration, applications would make use of the `registerTypeOverride` method of the `org.hibernate.cfg.Configuration` class when bootstrapping Hibernate. For example, lets say you want Hibernate to use your custom `SuperDuperStringType`; during bootstrap you would call:

#6.4 Overriding the standard `StringType`

```
Configuration cfg = ...;
cfg.registerTypeOverride( new SuperDuperStringType() );
```

The argument to `registerTypeOverride` is a `org.hibernate.type.BasicType` which is a specialization of the `org.hibernate.type.Type` we saw before. It adds a single method:

#6.5 Snippet from BasicType.java

```
/**
         * Get the names under which this type should be registered in the type registry.
         *
         * @return The keys under which to register this type.
         */
        public String[] getRegistrationKeys();
```

One approach is to use inheritance (`SuperDuperStringType` extends `org.hibernate.type.StringType`); another is to use delegation.

#############

7.1.

Naturally Hibernate also allows to persist collections. These persistent collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities. The distinction between value and reference semantics is in this context very important. An object in a collection might be handled with "value" semantics (its life cycle fully depends on the collection owner), or it might be a reference to another entity with its own life cycle. In the latter case, only the "link" between the two objects is considered to be a state held by the collection.

As a requirement persistent collection-valued fields must be declared as an interface type (see *#7.2#Collection mapping using @OneToMany and @JoinColumn#*). The actual interface might be `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` or anything you like ("anything you like" means you will have to write an implementation of `org.hibernate.usertype.UserCollectionType`).

Notice how in *#7.2#Collection mapping using @OneToMany and @JoinColumn#* the instance variable `parts` was initialized with an instance of `HashSet`. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent, by calling `persist()`, Hibernate will actually replace the `HashSet` with an instance of Hibernate's own implementation of `Set`. Be aware of the following error:

#7.1 Hibernate uses its own collection implementations

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);

kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

Hibernate ################################## `HashMap` # `HashSet`# `TreeMap`# `TreeSet`# `ArrayList` ##########

null ######## ########### Hibernate

##

Use persistent collections the same way you use ordinary Java collections. However, ensure you understand the semantics of bidirectional associations (see #######).

7.2. How to map collections

Using annotations you can map `Collections`, `Lists`, `Maps` and `Sets` of associated entities using @OneToMany and @ManyToMany. For collections of a basic or embeddable type use @ElementCollection. In the simplest case a collection mapping looks like this:

#7.2 Collection mapping using @OneToMany and @JoinColumn

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinColumn(name="PART_ID")
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


@Entity
public class Part {
   ...
}
```

Product describes a unidirectional relationship with Part using the join column PART_ID. In this unidirectional one to many scenario you can also use a join table as seen in *#7.3#Collection mapping using @OneToMany and @JoinTable#*.

#7.3 Collection mapping using @OneToMany and @JoinTable

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();
```

```
    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinTable(
            name="PRODUCT_PARTS",
            joinColumns = @JoinColumn( name="PRODUCT_ID"),
            inverseJoinColumns = @JoinColumn( name="PART_ID")
    )
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


@Entity
public class Part {
   ...
}
```

Without describing any physical mapping (no `@JoinColumn` or `@JoinTable`), a unidirectional one to many with join table is used. The table name is the concatenation of the owner table name, _, and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table, _, and the owner primary key column(s) name. The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s) name. A unique constraint is added to the foreign key referencing the other side table to reflect the one to many.

Lets have a look now how collections are mapped using Hibernate mapping files. In this case the first step is to chose the right mapping element. It depends on the type of interface. For example, a `<set>` element is used for mapping properties of type `Set`.

#7.4 Mapping a Set using <set>

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

In *#7.4#Mapping a Set using <set>#* a *one-to-many association* links the `Product` and `Part` entities. This association requires the existence of a foreign key column and possibly an index column to the `Part` table. This mapping loses certain semantics of normal Java collections:

- ###################2##############################
- ##2##############

Looking closer at the used `<one-to-many>` tag we see that it has the following options.

#7.5 options of <one-to-many> element

```
<one-to-many
        class="ClassName"                                    (1)
        not-found="ignore|exception"                         (2)
        entity-name="EntityName"                             (3)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) `class` ####: ##########

(2) `not-found` ###### - ###### exception#: ###################################: `ignore` ##############################

(3) `entity-name` #######: `class` ##################### `class` ######################## ##

`<one-to-many>` ################################ #### ################

##

If the foreign key column of a `<one-to-many>` association is declared `NOT NULL`, you must declare the `<key>` mapping `not-null="true"` or *use a bidirectional association* with the collection mapping marked `inverse="true"`. See ########.

Apart from the `<set>` tag as shown in *#7.4#Mapping a Set using <set>#*, there is also `<list>`, `<map>`, `<bag>`, `<array>` and `<primitive-array>` mapping elements. The `<map>` element is representative:

#7.6 Elements of the <map> mapping

```
<map
    name="propertyName"                                                    (1)
    table="table_name"                                                     (2)
    schema="schema_name"                                                   (3)
    lazy="true|extra|false"                                                (4)
    inverse="true|false"                                                   (5)
    cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"  (6)
    sort="unsorted|natural|comparatorClass"                                (7)
    order-by="column_name asc|desc"                                        (8)
    where="arbitrary sql where condition"                                  (9)
    fetch="join|select|subselect"                                          (10)
    batch-size="N"                                                         (11)
```

```
    access="field|property|ClassName"                                    (12)
    optimistic-lock="true|false"                                         (13)
    mutable="true|false"                                                 (14)
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

(1) name ##################

(2) table ###### - ##

(3) schema ##

(4) lazy ###### - ###### true## ##extra-lazy######################extra-lazy############################## ########## ##########

(5) inverse ###### - ###### false#################################

(6) cascade ###### - ###### none############################

(7) sort ############################## natural ########## Comparator ##########

(8) order-by (optional): specifies a table column or columns that define the iteration order of the Map, Set or bag, together with an optional asc or desc.

(9) where ############################## SQL #WHERE ######## ################## ####################

(10) fetch ###### - ###### select###################### #sequential select fetch# ######### #sequential subselect fetch# ##############

(11) batch-size ###### - ###### 1##

(12) access ###### - ###### property###################################

(13) optimistic-lock ###### - ###### true# ## ###################### ####################################

(14) mutable###### - ###### true# false ########################## ############## ##########

After exploring the basic mapping of collections in the preceding paragraphs we will now focus details like physical mapping considerations, indexed collections and collections of value types.

7.2.1.

On the database level collection instances are distinguished by the foreign key of the entity that owns the collection. This foreign key is referred to as the *collection key column*, or columns, of the collection table. The collection key column is mapped by the @JoinColumn annotation respectively the <key> XML element.

There can be a nullability constraint on the foreign key column. For most collections, this is implied. For unidirectional one-to-many associations, the foreign key column is nullable by default, so you may need to specify

```
@JoinColumn(nullable=false)
```

or

```
<key column="productSerialNumber" not-null="true"/>
```

The foreign key constraint can use `ON DELETE CASCADE`. In XML this can be expressed via:

```
<key column="productSerialNumber" on-delete="cascade"/>
```

In annotations the Hibernate specific annotation @OnDelete has to be used.

```
@OnDelete(action=OnDeleteAction.CASCADE)
```

See *#Key#* for more information about the `<key>` element.

7.2.2.

In the following paragraphs we have a closer at the indexed collections `List` and `Map` how the their index can be mapped in Hibernate.

7.2.2.1. Lists

Lists can be mapped in two different ways:

- as ordered lists, where the order is not materialized in the database
- as indexed lists, where the order is materialized in the database

To order lists in memory, add `@javax.persistence.OrderBy` to your property. This annotation takes as parameter a list of comma separated properties (of the target entity) and orders the collection accordingly (eg `firstname asc, age desc`), if the string is empty, the collection will be ordered by the primary key of the target entity.

#7.7 Ordered lists using `@OrderBy`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
```

```
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderBy("number")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

To store the index value in a dedicated column, use the `@javax.persistence.OrderColumn` annotation on your property. This annotations describes the column name and attributes of the column keeping the index value. This column is hosted on the table containing the association foreign key. If the column name is not specified, the default is the name of the referencing property, followed by underscore, followed by `ORDER` (in the following example, it would be `orders_ORDER`).

#7.8 Explicit index column using `@OrderColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderColumn(name="orders_index")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}
```

```
@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|--------------| |----------|
| Order        | | Customer |
|--------------| |----------|
| id           | | id       |
| number       | |----------|
| customer_id  |
| orders_order |
|--------------|
```

##

We recommend you to convert the legacy `@org.hibernate.annotations.IndexColumn` usages to `@OrderColumn` unless you are making use of the base property. The `base` property lets you define the index value of the first element (aka as base index). The usual value is `0` or `1`. The default is 0 like in Java.

Looking again at the Hibernate mapping file equivalent, the index of an array or list is always of type `integer` and is mapped using the `<list-index>` element. The mapped column contains sequential integers that are numbered from zero by default.

#7.9 index-list element for indexed collections in xml mapping

```
<list-index
        column="column_name"                                    (1)
        base="0|1|..."/>
```

(1) `column_name` (required): the name of the column holding the collection index values.

(1) `base` (optional - defaults to `0`): the value of the index column that corresponds to the first element of the list or array.

If your table does not have an index column, and you still wish to use `List` as the property type, you can map the property as a Hibernate <*bag*>. A bag does not retain its order when it is retrieved from the database, but it can be optionally sorted or ordered.

7.2.2.2. Maps

The question with `Maps` is where the key value is stored. There are everal options. Maps can borrow their keys from one of the associated entity properties or have dedicated columns to store an explicit key.

To use one of the target entity property as a key of the map, use `@MapKey(name="myProperty")`, where `myProperty` is a property name in the target entity. When using `@MapKey` without the name attribuate, the target entity primary key is used. The map key uses the same column as the property pointed out. There is no additional column defined to hold the map key, because the map key represent a target property. Be aware that once loaded, the key is no longer kept in sync with the property. In other words, if you change the property value, the key will not change automatically in your Java model.

#7.10 Use of target entity property as map key via `@MapKey`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @MapKey(name="number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> order) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
```

```
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

Alternatively the map key is mapped to a dedicated column or columns. In order to customize the mapping use one of the following annotations:

- `@MapKeyColumn` if the map key is a basic type. If you don't specify the column name, the name of the property followed by underscore followed by `KEY` is used (for example `orders_KEY`).
- `@MapKeyEnumerated` / `@MapKeyTemporal` if the map key type is respectively an enum or a `Date`.
- `@MapKeyJoinColumn`/`@MapKeyJoinColumns` if the map key type is another entity.
- `@AttributeOverride`/`@AttributeOverrides` when the map key is a embeddable object. Use `key.` as a prefix for your embeddable object property names.

You can also use `@MapKeyClass` to define the type of the key if you don't use generics.

#7.11 Map key as basic type using `@MapKeyColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany @JoinTable(name="Cust_Order")
   @MapKeyColumn(name="orders_number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> orders) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------| |---------------|
```

```
| Order       | | Customer | | Cust_Order    |
|-------------| |----------| |---------------|
| id          | | id       | | customer_id   |
| number      | |----------| | order_id      |
| customer_id |              | orders_number |
|-------------|              |---------------|
```

##

We recommend you to migrate from `@org.hibernate.annotations.MapKey` / `@org.hibernate.annotation.MapKeyManyToMany` to the new standard approach described above

Using Hibernate mapping files there exists equivalent concepts to the descibed annotations. You have to use `<map-key>`, `<map-key-many-to-many>` and `<composite-map-key>`. `<map-key>` is used for any basic type, `<map-key-many-to-many>` for an entity reference and `<composite-map-key>` for a composite type.

#7.12 map-key xml mapping element

```
<map-key
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="type_name"                        (3)
        node="@attribute-name"
        length="N"/>
```

(1) `column` (optional): the name of the column holding the collection index values.

(2) `formula` (optional): a SQL formula used to evaluate the key of the map.

(3) `type` (required): the type of the map keys.

#7.13 map-key-many-to-many

```
<map-key-many-to-many
        column="column_name"                    (1)
        formula="any SQL expression"            (2)(3)
        class="ClassName"
/>
```

(1) `column` (optional): the name of the foreign key column for the collection index values.

(2) `formula` (optional): a SQ formula used to evaluate the foreign key of the map key.

(3) `class` (required): the entity class used as the map key.

7.2.3. Collections of basic types and embeddable objects

In some situations you don't need to associate two entities but simply create a collection of basic types or embeddable objects. Use the `@ElementCollection` for this case.

#7.14 Collection of basic types mapped via `@ElementCollection`

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Nicknames", joinColumns=@JoinColumn(name="user_id"))
   @Column(name="nickname")
   public Set<String> getNicknames() { ... }
}
```

The collection table holding the collection data is set using the `@CollectionTable` annotation. If omitted the collection table name defaults to the concatenation of the name of the containing entity and the name of the collection attribute, separated by an underscore. In our example, it would be `User_nicknames`.

The column holding the basic type is set using the `@Column` annotation. If omitted, the column name defaults to the property name: in our example, it would be `nicknames`.

But you are not limited to basic types, the collection type can be any embeddable object. To override the columns of the embeddable object in the collection table, use the `@AttributeOverride` annotation.

#7.15 @ElementCollection for embeddable objects

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Addresses", joinColumns=@JoinColumn(name="user_id"))
   @AttributeOverrides({
      @AttributeOverride(name="street1", column=@Column(name="fld_street"))
   })
   public Set<Address> getAddresses() { ... }
}

@Embeddable
public class Address {
   public String getStreet1() {...}
   [...]
}
```

Such an embeddable object cannot contains a collection itself.

##

in `@AttributeOverride`, you must use the `value.` prefix to override properties of the embeddable object used in the map value and the `key.` prefix to override properties of the embeddable object used in the map key.

```
@Entity
public class User {
   @ElementCollection
   @AttributeOverrides({
      @AttributeOverride(name="key.street1", column=@Column(name="fld_street")),
      @AttributeOverride(name="value.stars", column=@Column(name="fld_note"))
   })
   public Map<Address,Rating> getFavHomes() { ... }
```

##

We recommend you to migrate from `@org.hibernate.annotations.CollectionOfElements` to the new `@ElementCollection` annotation.

Using the mapping file approach a collection of values is mapped using the `<element>` tag. For example:

#7.16 <element> tag for collection values using mapping files

```
<element
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="typename"                         (3)
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

1. `column` (optional): the name of the column holding the collection element values.
2. `formula` (optional): an SQL formula used to evaluate the element.
3. `type` (required): the type of the collection element.

7.3.

7.3.1.

Hibernate supports collections implementing `java.util.SortedMap` and `java.util.SortedSet`. With annotations you declare a sort comparator using `@Sort`. You chose between the comparator types unsorted, natural or custom. If you want to use your own comparator implementation, you'll also have to specify the implementation class using the `comparator` attribute. Note that you need to use either a `SortedSet` or a `SortedMap` interface.

#7.17 Sorted collection with @Sort

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

Using Hibernate mapping files you specify a comparator in the mapping file with `<sort>`:

#7.18 Sorted collection using xml mapping

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

`sort` ########## `unsorted` # `natural` #### `java.util.Comparator` ###############

####

################## `java.util.TreeSet` # `java.util.TreeMap` ##########

If you want the database itself to order the collection elements, use the `order-by` attribute of `set`, `bag` or `map` mappings. This solution is implemented using `LinkedHashSet` or `LinkedHashMap` and performs the ordering in the SQL query and not in the memory.

#7.19 Sorting in database using order-by

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

##

The value of the `order-by` attribute is an SQL ordering, not an HQL ordering.

########### `filter()` ############### criteria ##################

#7.20 Sorting via a query filter

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

7.3.2.

############################2####################

one-to-many

set # bag

many-to-many

set # bag

Often there exists a many to one association which is the owner side of a bidirectional relationship. The corresponding one to many association is in this case annotated by `@OneToMany(mappedBy=...)`

#7.21 Bidirectional one to many with many to one side as association owner

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}
```

```
@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

`Troop` has a bidirectional one to many relationship with `Soldier` through the `troop` property. You don't have to (must not) define any physical mapping in the `mappedBy` side.

To map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the `mappedBy` element and set the many to one `@JoinColumn` as insertable and updatable to false. This solution is not optimized and will produce additional UPDATE statements.

#7.22 Bidirectional associtaion with one to many side as owner

```
@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

How does the mappping of a bidirectional mapping look like in Hibernate mapping xml? There you define a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end `inverse="true"`.

#7.23 Bidirectional one to many via Hibernate mapping files

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
```

```
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

inverse="true"

A many-to-many association is defined logically using the `@ManyToMany` annotation. You also have to describe the association table and the join conditions using the `@JoinTable` annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

#7.24 Many to many association via @ManyToMany

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

In this example `@JoinTable` defines a `name`, an array of join columns, and an array of inverse join columns. The latter ones are the columns of the association table which refer to the `Employee` primary key (the "other side"). As seen previously, the other side don't have to (must not) describe the physical mapping: a simple `mappedBy` argument containing the owner side property name bind the two.

As any other annotations, most values are guessed in a many to many relationship. Without describing any physical mapping in a unidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table name, _ and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

#7.25 Default values for `@ManyToMany` (uni-directional)

```
@Entity
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}
```

A `Store_City` is used as the join table. The `Store_id` column is a foreign key to the `Store` table. The `implantedIn_id` column is a foreign key to the `City` table.

Without describing any physical mapping in a bidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, _, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

#7.26 Default values for `@ManyToMany` (bi-directional)

```
@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
```

```
}
```

A `Store_Customer` is used as the join table. The `stores_id` column is a foreign key to the `Store` table. The `customers_id` column is a foreign key to the `Customer` table.

Using Hibernate mapping files you can map a bidirectional many-to-many association by mapping two many-to-many associations to the same database table and declaring one end as *inverse*.

##

You cannot select an indexed collection.

#7.27#Many to many association using Hibernate mapping files# shows a bidirectional many-to-many association that illustrates how each category can have many items and each item can be in many categories:

#7.27 Many to many association using Hibernate mapping files

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="ITEM_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

inverse ############## ########## Hibernate ################################## ########## A ## B ########## B ## A ############## Java ################## Java # ###################################### Java

#7.28 Effect of inverse vs. non-inverse side of many to many associations

```
category.getItems().add(item);           // The category now "knows" about the relationship
item.getCategories().add(category);      // The item now "knows" about the relationship

session.persist(item);                   // The relationship won't be saved!
```

```
session.persist(category);              // The relationship will be saved
```

inverse

7.3.3.

There are some additional considerations for bidirectional mappings with indexed collections (where one end is represented as a `<list>` or `<map>`) when using Hibernate mapping files. If there is a property of the child class that maps to the index column you can use `inverse="true"` on the collection mapping:

#7.29 Bidirectional association with indexed collection

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

############################
#################### `inverse="true"` ###################################

#7.30 Bidirectional association with indexed collection, but no index column

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>
```

```
<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

Note that in this mapping, the collection-valued end of the association is responsible for updates to the foreign key.

7.3.4. 3###

3###########3##############1################ `Map` ##############

#7.31 Ternary association mapping

```
@Entity
public class Company {
   @Id
   int id;
   ...
   @OneToMany // unidirectional
   @MapKeyJoinColumn(name="employee_id")
   Map<Employee, Contract> contracts;
}

// or

<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

A second approach is to remodel the association as an entity class. This is the most common approach. A final alternative is to use composite elements, which will be discussed later.

7.3.5. `Using an <idbag>`

The majority of the many-to-many associations and collections of values shown previously all map to tables with composite keys, even though it has been suggested that entities should have synthetic identifiers (surrogate keys). A pure association table does not seem to benefit much from a surrogate key, although a collection of composite values *might*. For this reason Hibernate provides a feature that allows you to map many-to-many associations and collections of values to a table with a surrogate key.

bag ############## List#### Collection## <idbag> ##############

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

######## <idbag> ################### id ## ###### Hibernate ##############################

<idbag> ################ <bag> ################## Hibernate ##################### list # map # set ######################

######### native ### id ##### <idbag> ####################

7.4.

This section covers collection examples.

The following class has a collection of Child instances:

#7.32 Example classes Parent and Child

```
public class Parent {
    private long id;
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   private long id;
   private String name


   // getter/setter
   ...
}
```

If each child has, at most, one parent, the most natural mapping is a one-to-many association:

#7.33 One to many unidirectional `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
}
```

#7.34 One to many unidirectional `Parent-Child` relationship using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

##########################

#7.35 Table definitions for unidirectional `Parent-Child` relationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

parent # ##

#7.36 One to many bidirectional `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy="parent")
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   @ManyToOne
   private Parent parent;


   // getter/setter
   ...
}
```

#7.37 One to many bidirectional `Parent-Child` relationship using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
```

```
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

NOT NULL ##############

#7.38 Table definitions for bidirectional `Parent-Child` relationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Alternatively, if this association must be unidirectional you can enforce the `NOT NULL` constraint.

#7.39 Enforcing NOT NULL constraint in unidirectional relation using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(optional=false)
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
}
```

#7.40 Enforcing NOT NULL constraint in unidirectional relation using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

On the other hand, if a child has multiple parents, a many-to-many association is appropriate.

#7.41 Many to many `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @ManyToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;


   // getter/setter
   ...
}
```

#7.42 Many to many `Parent-Child` relationship using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

##################:

#7.43 Table definitions for many to many releationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

For more examples and a complete explanation of a parent/child relationship mapping, see *24 ### #/##* for more information. Even more complex association mappings are covered in the next chapter.

#######

8.1.

##
###################### Person # Address ######

##

null ## not null ################ Hibernate ############ not null ############################

8.2.

8.2.1. Many-to-one

######## ####################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

8.2.2. One-to-one

############## ###################################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
```

```
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

############# ######## ID ############# ##################################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property"
>person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

8.2.3. One-to-many

############## ##########################

```
<class name="Person">
    <id name="id" column="personId">
```

```
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

##############################

8.3.

8.3.1. One-to-many

################## ########## unique="true" ##################################
##

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```

```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

8.3.2. Many-to-one

############### ########################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

8.3.3. One-to-one

################ ######################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
```

```
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
 unique )
create table Address ( addressId bigint not null primary key )
```

8.3.4. Many-to-many

######

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.4.

8.4.1. ###/###

######## ############ ############

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

List ################################## key #### not null ################## ############################## #update="false" ## insert="false" ############## inverse ######

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
```

```
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class
>
```

If the underlying foreign key column is `NOT NULL`, it is important that you define `not-null="true"` on the `<key>` element of the collection mapping. Do not only declare `not-null="true"` on a possible nested `<column>` element, but on the `<key>` element.

8.4.2. One-to-one

############## ##########

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

############# #### ID #############

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
```

```
            <param name="property"
>person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

8.5.

8.5.1. ###/###

################ ### inverse="true" ######################################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class
>
```

```
create table Person ( personId bigint not null primary key )
```

```
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

8.5.2.

################ ################

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
 unique )
create table Address ( addressId bigint not null primary key )
```

8.5.3. Many-to-many

######

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.6.

########## ### ################ SQL ################################ accountNumber # effectiveEndDate # effectiveStartDate ###### account ############ ######################

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula
>case when effectiveEndDate is null then 1 else 0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

######## #### ###### #effectiveEndDate # null ############################

```
<many-to-one name="currentAccountInfo"
```

```
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula
>'1'</formula>
</many-to-one
>
```

########## Employee##### # Organization#### ##### Employment#### ############### ## ##### ######## ###### startDate ###### ##################

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join
>
```

############################ HQL # criteria ################

##########

####### ##### Hibernate ############################

9.1.

##
Person

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
```

```
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

`Name` # `Person` ############################ `Name` ########## getter # setter #######

####################

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"
> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class
>
```

Person ##### `pid`# `birthday`# `initial`# `first`# `last` #########

Person ##################### Person ################### name ############################# null ########## #### # ############################ Hibernate ################# null ################ null

############## Hibernate ################# many-to-one ########################### ################### ##### # Hibernate ###

`<component>` ### `<parent>` ###########

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class
>
```

9.2.

Hibernate ############################## Name ###### <element> ### <composite-element> ####################################

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"
> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set
>
```

##: ############ Set ######## equals() # hashCode() ##################

<nested-composite-element> ## ############ one-to-many ## ############# Java

<set> ############################## null ############################## Hibernate ## null ########## ## ################################### ########### not-null ############### <list>#<map># <bag>#<idbag>

############################## <many-to-one> ################################ ## Order ###Item ########## purchaseDate# price# quantity ############

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
        </composite-element>
    </set>
</class
>
```

############################ purchase ## ############### Purchase #### Order # set ############ Item ###############

3########4##########

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class
>
```

##

9.3. Map

<composite-map-key> ### Map ## hashCode() # equals() ##################

9.4.

###

- java.io.Serializable ##############
- ############################ equals() # hashCode() ###############

##

Hibernate3 #####2##

############ IdentifierGenerator ##

<id> ####### <composite-id> ### ####### <key-property> ###### ############ OrderLine #### Order

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>
```

```
    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class
>
```

`OrderLine` ## `OrderLine`

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one
>
```

####

The `column` element is an alternative to the `column` attribute everywhere. Using the `column` element just gives more declaration options, which are mostly useful when utilizing `hbm2ddl`

`OrderLine` ## `many-to-many` ###############

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set
>
```

`Order` ### `OrderLine` ###################

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
```

```
    <one-to-many class="OrderLine"/>
</set
>
```

#<one-to-many> ####################

OrderLine ####################################

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key
>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class
>
```

9.5.

Map ###################

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component
>
```

<dynamic-component> ############## <component> ################################ ############## Bean ################ DOM ###################################### ## Configuration ######### Hibernate ####################################

#######

10.1. 3####

Hibernate #3#########################

- ############ #table-per-class-hierarchy#
- table per subclass
- ############ #table-per-concrete-class#

###4#### Hibernate ################################

- ############

###
################## Hibernate ###### `<class>` #### `<subclass>` ###### `<joined-subclass>` ###### `<union-subclass>` ###################### `<subclass>` ### `<join>` ########## #### `<class>` ##### table-per-hierarchy ### table-per-subclass ########################

`subclass# union-subclass # joined-subclass ################################ hibernate-mapping ## ################################ extends ### ########## Hibernate3 #### extends ### ##)`

```
 <hibernate-mapping>
     <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
          <property name="name" type="string"/>
     </subclass>
 </hibernate-mapping
>
```

10.1.1. ##############table-per-class-hierarchy#

############ `Payment` ######### `CreditCardPayment# CashPayment# ChequePayment` #### ##############################:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
```

```
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class
>
```

CCTYPE ################### NOT NULL

10.1.2. ############ #table-per-subclass#

table-per-subclass #################:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class
>
```

#########4#############3## ############ #######

10.1.3. discriminator #### table-per-subclass

Hibernate # table-per-subclass #### discriminator ######################## Hibernate ### O/R ###### table-per-subclass ################################## discriminator ######## ## table-per-subclass ### discriminator ## ######### <subclass> # <join> ####################

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class
>
```

`fetch="select"` ################################## `ChequePayment` ########

10.1.4. table-per-subclass # table-per-class-hierarchy

############### table-per-hierarchy # table-per-subclass ################

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
```

```
    </subclass>
</class
>
```

########################## Payment ################## <many-to-one> ##############

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

10.1.5. ###############table-per-concrete-class#

table-per-concrete-class ####################2######1### <union-subclass> ##########

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class
>
```

########3###

The limitation of this approach is that if a property is mapped on the superclass, the column name must be the same on all subclass tables. The identity generator strategy is not allowed in union subclass inheritance. The primary key seed has to be shared across all unioned subclasses of a hierarchy.

################### abstract="true" ## ############################## (############ PAYMENT)#

10.1.6. ################ table-per-concrete-class

##############################

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
```

```
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class
>
```

Payment ## Payment ################### ################################## XML ################## ### DOCTYPE #### ## [<!ENTITY allproperties SYSTEM "allproperties.xml">] ########## &allproperties;##

############ Hibernate ################## SQL UNION ##########

############### Payment ################## <any> ##############

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any
>
```

10.1.7.

################################## <class> ################### Payment ###### ##Payment ################ ######################

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
```

```
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class
>
```

########## Payment ################ Payment ###################### #### from Payment ####### Hibernate ##### CreditCardPayment ## CreditCardPayment # ###### Payment ############## CashPayment # ChequePayment ############# NonelectronicTransaction #############

10.2.

table-per-concrete-class ## <union-subclass> ######################

The following table shows the limitations of table per concrete-class mappings, and of implicit polymorphism, in Hibernate.

#10.1

####	Polymorp many-to-one	###### ######	###### ######	###### ######	#### ##### load()/get()	###### ###### ###	####### #######	Outer join fetching
table per class-hierarchy	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	fr om Order o join o.payment p	*supported*

####	Polymorp many-to-one	###### ######	###### ######	###### ######	#### ##### load()/ get()	###### ###### ###	####### #######	Outer join fetching
table per subclass	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	so Order o join o.payment p	*supported*
table per concrete-class (union-subclass)	<many-to-one>	<one-to-one>	<one-to-many> (for inverse="true" only)	<many-to-many>	s.get(Payment.class, id)	from Payment p	so Order o join o.payment p	*supported*
table per concrete class (implicit polymorphism)	<any>	*not supported*	*not supported*	<many-to-any>	s.createCriteria(Payment.class).add(Restrictions.	Criteria p	*not supported*	*not supported*

##########

Hibernate ##########/## ##### #### ########## JDBC/SQL ######### SQL `statements` ######### Java ######### ###################################

####### Hibernate ######################### ## ################ SQL ############ ################### Hibernate ##

11.1. Hibernate

Hibernate ##############################:

- *Transient* - `new` ###################### Hibernate # `Session` ################## ## transient ## Transient ########## ## (persistent) ########### Hibernate # `Session` ################### SQL ###### Hibernate #########
- *### (Persistent)* - ## ################## `Session` ################ Hibernate #######Unit of Work####### ## transient ########## ###### `UPDATE` ## `DELETE` #########
- *Detached* - detached ############################## `Session` ############## ##detached ######################### detached ## `Session` ############## ## ############## *#####application transactions#* ###################################

############################ Hibernate ###################

11.2.

Newly instantiated instances of a persistent class are considered *transient* by Hibernate. We can make a transient instance *persistent* by associating it with a session:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

`Cat` ##################### `save()` ################## `cat` ################ `Cat` # ##### `assigned` ################## `save()` ############ `cat` ##################### `save()` ###### EJB3 ############## `persist()` ############

- `persist()` makes a transient instance persistent. However, it does not guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time. `persist()` also guarantees that it will not execute an `INSERT` statement if it is called outside of transaction boundaries. This is useful in long-running conversations with an extended Session/persistence context.
- `save()` does guarantee to return an identifier. If an INSERT has to be executed to get the identifier (e.g. "identity" generator, not "sequence"), this INSERT happens immediately, no matter if you are inside or outside of a transaction. This is problematic in a long-running conversation with an extended Session/persistence context.

Alternatively, you can assign the identifier using an overloaded version of `save()`.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

############################## ############ `kittens` ###################### `NOT NULL` ## `NOT NULL` ######################## `save()` ########################

###################### Hibernate # ####### *(transitive persistence)* ###################### ################## `NOT NULL` ############### Hibernate ############################## ##########

11.3.

################################## `Session` # `load()` ################ `load()` ## Class ## (persistent) #####

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

##:

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
```

```
Set kittens = cat.getKittens();
```

DB ############ `load()` ### `load()` ## ## `batch-size` ########################## #######################

######################## `get()` ## null # #####

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

`LockMode` ##### `SELECT ... FOR UPDATE` ### SQL ################################ API #################

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

################# `lock` # `all` ################################ `FOR UPDATE` ### #### # ##########

`refresh()` ## ##

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

How much does Hibernate load from the database and how many SQL SELECTs will it use? This depends on the *fetching strategy*. This is explained in ########.

11.4.

###################################### Hibernate ######################### (HQL) # ################################## Hibernate ###### Criteria # Example ##### (QBC # QBE# ########### ResultSet ############ Hibernate ############################# SQL #################

11.4.1.

HQL ####### SQL ##### `org.hibernate.Query` ##################################### ##### ResultSet ################################### `Query` ################ `Session` ##########

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

######## `list()` ## ###################################1#################### `uniqueResult()` ##### ### ############## (################## Set ################

11.4.1.1.

`iterate()` ##
`iterate()` ## `list()` ## ##### select ## ######################### *n*# # select ######

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
```

```
        break;
    }
}
```

11.4.1.2. #########tuple#######

Hibernate ##:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = (Cat) tuple[0];
    Cat mother = (Cat) tuple[1];
    ....
}
```

11.4.1.3.

`select` ################### SQL ##

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

11.4.1.4.

`Query` ############ JDBC ##### `?` ############################## *JDBC* ##### *Hibernate* ## `:name` ############### ##############

- ################################
- ######################

- ##########

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

11.4.1.5.

ResultSet ## `Query` ###################

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

######### DBMS ######## SQL ######### Hibernate ########

11.4.1.6.

JDBC ############## `ResultSet` ############ `Query` ############## `ScrollableResults` # ##

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
```

```
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
cats.close()
```

##
`setMaxResult()` / `setFirstResult()` ########

11.4.1.7.

Queries can also be configured as so called named queries using annotations or Hibernate mapping documents. `@NamedQuery` and `@NamedQueries` can be defined at the class level as seen in *#11.1#Defining a named query using @NamedQuery#* . However their definitions are global to the session factory/entity manager factory scope. A named query is defined by its name and the actual query string.

#11.1 Defining a named query using `@NamedQuery`

```
@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}
```

Using a mapping document can be configured using the `<query>` node. Remember to use a `CDATA` section if your query contains characters that could be interpreted as markup.

#11.2 Defining a named query using `<query>`

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
```

```
] ]></query>
```

Parameter binding and executing is done programatically as seen in *#11.3#Parameter binding of a named query#*.

#11.3 Parameter binding of a named query

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

SQL ################## ########################## Hibernate

<hibernate-mapping> ## <class> ##
eg.Cat.ByNameAndMaximumWeight

11.4.2.

this

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

############ Bag ## "filter" # ###################################

from

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

##

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
```

```
    .setFirstResult(0).setMaxResults(10)
    .list();
```

11.4.3.

HQL ################################ API ################################## Hibernate ##### `Criteria` ### API #########

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

The `Criteria` and the associated `Example` API are discussed in more detail in *17#Criteria ###*.

11.4.4. ##### SQL

`createSQLQuery()` ##### SQL ##################### Hibernate ## ResultSet ########## ########### `session.connection()` ############### JDBC `Connection` ############ ######### Hibernate API ################ SQL ####################

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
.list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
.list()
```

SQL queries can contain named and positional parameters, just like Hibernate queries. More information about native SQL queries in Hibernate can be found in *18###### SQL*.

11.5.

########### ### `Session` ## ########## `Session` # ##### ## #### `update()` ## `load()` ## `Session` ########################

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
```

```
sess.flush();  // changes to cat are automatically detected and persisted
```

################# SQL # `SELECT` ################### SQL # `UPDATE` ############### ##################################### Hibernate ################## detached ##### ###########

11.6. detached

UI ############ ##

Hibernate ## `Session.update()` # `Session.merge()` ######### detached ################# #############

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

`catId` ### `Cat` #### `secondSession`

`update()` ########################### ################## `merge()` ########### detached ################################ ###### `update()`

The application should individually `update()` detached instances that are reachable from the given detached instance *only* if it wants their state to be updated. This can be automated using *transitive persistence*. See ######### for more information.

`lock()` ## detached #####################

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

`lock()` ####### `LockMode` ################## API ################################# ####### `lock()` ###############

Other models for long units of work are discussed in ##########.

11.7.

Hibernate ########2## transient ################################## detached #########/################## `saveOrUpdate()` ####################

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mate has a null id)
```

`saveOrUpdate()` ## `update()` # `saveOrUpdate()` # `merge()` ########################## ##########################

`update()` # `saveOrUpdate()` ##############:

- ##################################
- ######## UI ########
- ####################
- ########################
- ##########2######### `update()` ####################

`saveOrUpdate()` ###########:

- ####################################
- ##
- ###################### `save()` ####
- ## `save()` ####
- ######## `<version>` # `<timestamp>` ## ################################## `save()` ####
- ################## `update()` ####

`merge()` ###############:

- ## ##
- ##
- ################
- ##

11.8.

Session.delete() ##
######## delete() ########## transient #################

```
sess.delete(cat);
```

##
NOT NULL ###

11.9.

##

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

########################## replicate() ############## ReplicationMode #######

- ReplicationMode.IGNORE - ######################################
- ReplicationMode.OVERWRITE - ########################
- ReplicationMode.EXCEPTION - ################################
- ReplicationMode.LATEST_VERSION - ##
  ##########

ACID
################################

11.10.

JDBC ################################## SQL ## Session ####################
flush ####################

- #########
- org.hibernate.Transaction.commit() #######
- Session.flush() #######

SQL ###############

1. ################## `Session.save()` ############################
2. #############
3. #############
4. ########################
5. #############
6. ################## `Session.delete()` ############################

(########## `native` ID ###################################

`flush()` ######### ## `Session` # JDBC ## ## ############# Hibernate ## `Query.list(..)`

It is possible to change the default behavior so that flush occurs less frequently. The `FlushMode` class defines three different modes: only flush at commit time when the Hibernate `Transaction` API is used, flush automatically using the explained routine, or never flush unless `flush()` is called explicitly. The last mode is useful for long running units of work, where a `Session` is kept open and disconnected for a long time (see ####################).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

During flush, an exception might occur (e.g. if a DML operation violates a constraint). Since handling exceptions involves some understanding of Hibernate's transactional behavior, we discuss it in *13#Transactions and Concurrency*.

11.11.

##############################:

##################1############################## Hibernate ######################

##
##
Hibernate #
###############

Hibernate # Session ###### `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()` ##
####### `create`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate` ###
##:

```
<one-to-one name="person" cascade="persist"/>
```

########################:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

`cascade="all"` ############ `cascade="none"`
##################

In case you are using annotatons you probably have noticed the `cascade` attribute taking an array of `CascadeType` as a value. The cascade concept in JPA is very is similar to the transitive persistence and cascading of operations as described above, but with slightly different semantics and cascading types:

- `CascadeType.PERSIST`: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed
- `CascadeType.MERGE`: cascades the merge operation to associated entities if merge() is called or if the entity is managed
- `CascadeType.REMOVE`: cascades the remove operation to associated entities if delete() is called
- `CascadeType.REFRESH`: cascades the refresh operation to associated entities if refresh() is called
- `CascadeType.DETACH`: cascades the detach operation to associated entities if detach() is called
- `CascadeType.ALL`: all of the above

##

CascadeType.ALL also covers Hibernate specific operations like save-update, lock etc...

A special cascade style, `delete-orphan`, applies only to one-to-many associations, and indicates that the `delete()` operation should be applied to any child object that is removed from the association. Using annotations there is no `CascadeType.DELETE-ORPHAN` equivalent. Instead you can use the attribute `orphanRemoval as seen in` *#11.4#@OneToMany with orphanRemoval#*. If an entity is removed from a `@OneToMany` collection or an associated entity is dereferenced from a `@OneToOne` association, this associated entity can be marked for deletion if `orphanRemoval` is set to true.

#11.4 `@OneToMany` with `orphanRemoval`

```
@Entity
public class Customer {
   private Set<Order> orders;

   @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)
   public Set<Order> getOrders() { return orders; }

   public void setOrders(Set<Order> orders) { this.orders = orders; }

   [...]
}

@Entity
public class Order { ... }

Customer customer = em.find(Customer.class, 1l);
Order order = em.find(Order.class, 1l);
customer.getOrders().remove(order); //order will be deleted by cascade
```

#####

- It does not usually make sense to enable cascade on a many-to-one or many-to-many association. In fact the `@ManyToOne` and `@ManyToMany` don't even offer a `orphanRemoval` attribute. Cascading is often useful for one-to-one and one-to-many associations.
- If the child object's lifespan is bounded by the lifespan of the parent object, make it a *life cycle object* by specifying `cascade="all,delete-orphan"(@OneToMany(cascade=CascadeType.ALL, orphanRemoval=true))`.
- ## ################ `cascade="persist,merge,save-update"` ##############

`cascade="all"` ########################## ## ##########################/##/###### ###/##/#################

Furthermore, a mere reference to a child from a persistent parent will result in save/update of the child. This metaphor is incomplete, however. A child which becomes unreferenced by its parent is *not* automatically deleted, except in the case of a one-to-many association mapped with `cascade="delete-orphan"`. The precise semantics of cascading operations for a parent/child relationship are as follows:

- ## persist() ################ persist() #######
- merge() ################ merge() #######
- ## save() # update() # saveOrUpdate() ################ saveOrUpdate() #######
- transient ### detached ###################### saveOrUpdate() #######
- ################## delete() #######
- ########################## ########## ## ## cascade="delete-orphan" ##############################

################################ ###### ##### *flush*### ######################## ### save-upate # delete-orphan ## Session # flush ##################################

11.12.

Hibernate ## ################### Hibernate ## ## #########

Hibernate # ClassMetadata # CollectionMetadata ######## Type ##################### ##################### SessionFactory ########

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Read-only entities

####

Hibernate's treatment of *read-only* entities may differ from what you may have encountered elsewhere. Incorrect usage may cause unexpected results.

When an entity is read-only:

- Hibernate does not dirty-check the entity's simple properties or single-ended associations;
- Hibernate will not update simple properties or updatable single-ended associations;
- Hibernate will not update the version of the read-only entity if only simple properties or single-ended updatable associations are changed;

In some ways, Hibernate treats read-only entities the same as entities that are not read-only:

- Hibernate cascades operations to associations as defined in the entity mapping.
- Hibernate updates the version if the entity has a collection with changes that dirties the entity;
- A read-only entity can be deleted.

Even if an entity is not read-only, its collection association can be affected if it contains a read-only entity.

For details about the affect of read-only entities on different property and association types, see *#Read-only affect on property type#*.

For details about how to make entities read-only, see *#Making persistent entities read-only#*

Hibernate does some optimizing for read-only entities:

- It saves execution time by not dirty-checking simple properties or single-ended associations.
- It saves memory by deleting database snapshots.

12.1. Making persistent entities read-only

Only persistent entities can be made read-only. Transient and detached entities must be put in persistent state before they can be made read-only.

Hibernate provides the following ways to make persistent entities read-only:

- you can map an entity class as *immutable*; when an entity of an immutable class is made persistent, Hibernate automatically makes it read-only. see *#Entities of immutable classes#* for details
- you can change a default so that entities loaded into the session by Hibernate are automatically made read-only; see *#Loading persistent entities as read-only#* for details
- you can make an HQL query or criteria read-only so that entities loaded when the query or criteria executes, scrolls, or iterates, are automatically made read-only; see *#Loading read-only entities from an HQL query/criteria#* for details
- you can make a persistent entity that is already in the in the session read-only; see *#Making a persistent entity read-only#* for details

12.1.1. Entities of immutable classes

When an entity instance of an immutable class is made persistent, Hibernate automatically makes it read-only.

An entity of an immutable class can created and deleted the same as an entity of a mutable class.

Hibernate treats a persistent entity of an immutable class the same way as a read-only persistent entity of a mutable class. The only exception is that Hibernate will not allow an entity of an immutable class to be changed so it is not read-only.

12.1.2. Loading persistent entities as read-only

##

Entities of immutable classes are automatically loaded as read-only.

To change the default behavior so Hibernate loads entity instances of mutable classes into the session and automatically makes them read-only, call:

```
Session.setDefaultReadOnly( true );
```

To change the default back so entities loaded by Hibernate are not made read-only, call:

```
Session.setDefaultReadOnly( false );
```

You can determine the current setting by calling:

```
Session.isDefaultReadOnly();
```

If Session.isDefaultReadOnly() returns true, entities loaded by the following are automatically made read-only:

- Session.load()
- Session.get()
- Session.merge()
- executing, scrolling, or iterating HQL queries and criteria; to override this setting for a particular HQL query or criteria see *#Loading read-only entities from an HQL query/criteria#*

Changing this default has no effect on:

- persistent entities already in the session when the default was changed
- persistent entities that are refreshed via Session.refresh(); a refreshed persistent entity will only be read-only if it was read-only before refreshing
- persistent entities added by the application via Session.persist(), Session.save(), and Session.update() Session.saveOrUpdate()

12.1.3. Loading read-only entities from an HQL query/criteria

##

Entities of immutable classes are automatically loaded as read-only.

If Session.isDefaultReadOnly() returns false (the default) when an HQL query or criteria executes, then entities and proxies of mutable classes loaded by the query will not be read-only.

You can override this behavior so that entities and proxies loaded by an HQL query or criteria are automatically made read-only.

For an HQL query, call:

```
Query.setReadOnly( true );
```

`Query.setReadOnly( true )` must be called before `Query.list()`, `Query.uniqueResult()`, `Query.scroll()`, or `Query.iterate()`

For an HQL criteria, call:

```
Criteria.setReadOnly( true );
```

`Criteria.setReadOnly( true )` must be called before `Criteria.list()`, `Criteria.uniqueResult()`, or `Criteria.scroll()`

Entities and proxies that exist in the session before being returned by an HQL query or criteria are not affected.

Uninitialized persistent collections returned by the query are not affected. Later, when the collection is initialized, entities loaded into the session will be read-only if Session.isDefaultReadOnly() returns true.

Using `Query.setReadOnly( true )` or `Criteria.setReadOnly( true )` works well when a single HQL query or criteria loads all the entities and intializes all the proxies and collections that the application needs to be read-only.

When it is not possible to load and initialize all necessary entities in a single query or criteria, you can temporarily change the session default to load entities as read-only before the query is executed. Then you can explicitly initialize proxies and collections before restoring the session default.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

setDefaultReadOnly( true );
Contract contract =
   ( Contract ) session.createQuery(
           "from Contract where customerName = 'Sherman'" )
           .uniqueResult();
Hibernate.initialize( contract.getPlan() );
Hibernate.initialize( contract.getVariations() );
Hibernate.initialize( contract.getNotes() );
setDefaultReadOnly( false );
...
tx.commit();
session.close();
```

If Session.isDefaultReadOnly() returns true, then you can use Query.setReadOnly(false) and Criteria.setReadOnly(false) to override this session setting and load entities that are not read-only.

12.1.4. Making a persistent entity read-only

##

Persistent entities of immutable classes are automatically made read-only.

To make a persistent entity or proxy read-only, call:

```
Session.setReadOnly(entityOrProxy, true)
```

To change a read-only entity or proxy of a mutable class so it is no longer read-only, call:

```
Session.setReadOnly(entityOrProxy, false)
```

####

When a read-only entity or proxy is changed so it is no longer read-only, Hibernate assumes that the current state of the read-only entity is consistent with its database representation. If this is not true, then any non-flushed changes made before or while the entity was read-only, will be ignored.

To throw away non-flushed changes and make the persistent entity consistent with its database representation, call:

```
session.refresh( entity );
```

To flush changes made before or while the entity was read-only and make the database representation consistent with the current state of the persistent entity:

```
// evict the read-only entity so it is detached
session.evict( entity );

// make the detached entity (with the non-flushed changes) persistent
session.update( entity );

// now entity is no longer read-only and its changes can be flushed
s.flush();
```

12.2. Read-only affect on property type

The following table summarizes how different property types are affected by making an entity read-only.

#12.1 Affect of read-only entity on property types

Property/Association Type	Changes flushed to DB?
Simple	no*

Property/Association Type	Changes flushed to DB?
(*#Simple properties#*)	
Unidirectional one-to-one Unidirectional many-to-one (*#Unidirectional one-to-one and many-to-one#*)	no* no*
Unidirectional one-to-many Unidirectional many-to-many (*#Unidirectional one-to-many and many-to-many#*)	yes yes
Bidirectional one-to-one (*#Bidirectional one-to-one#*)	only if the owning entity is not read-only*
Bidirectional one-to-many/many-to-one inverse collection non-inverse collection (*#Bidirectional one-to-many/many-to-one#*)	only added/removed entities that are not read-only* yes
Bidirectional many-to-many (*#Bidirectional many-to-many#*)	yes

* Behavior is different when the entity having the property/association is read-only, compared to when it is not read-only.

12.2.1. Simple properties

When a persistent object is read-only, Hibernate does not dirty-check simple properties.

Hibernate will not synchronize simple property state changes to the database. If you have automatic versioning, Hibernate will not increment the version if any simple properties change.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// get a contract and make it read-only
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );

// contract.getCustomerName() is "Sherman"
contract.setCustomerName( "Yogi" );
tx.commit();
```

```
tx = session.beginTransaction();

contract = ( Contract ) session.get( Contract.class, contractId );
// contract.getCustomerName() is still "Sherman"
...
tx.commit();
session.close();
```

12.2.2. Unidirectional associations

12.2.2.1. Unidirectional one-to-one and many-to-one

Hibernate treats unidirectional one-to-one and many-to-one associations in the same way when the owning entity is read-only.

We use the term *unidirectional single-ended association* when referring to functionality that is common to unidirectional one-to-one and many-to-one associations.

Hibernate does not dirty-check unidirectional single-ended associations when the owning entity is read-only.

If you change a read-only entity's reference to a unidirectional single-ended association to null, or to refer to a different entity, that change will not be flushed to the database.

##

If an entity is of an immutable class, then its references to unidirectional single-ended associations must be assigned when that entity is first created. Because the entity is automatically made read-only, these references can not be updated.

If automatic versioning is used, Hibernate will not increment the version due to local changes to unidirectional single-ended associations.

In the following examples, Contract has a unidirectional many-to-one association with Plan. Contract cascades save and update operations to the association.

The following shows that changing a read-only entity's many-to-one association reference to null has no effect on the entity's database representation.

```
// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
contract.setPlan( null );
tx.commit();

// get the same contract
```

```
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );

// contract.getPlan() still refers to the original plan;

tx.commit();
session.close();
```

The following shows that, even though an update to a read-only entity's many-to-one association has no affect on the entity's database representation, flush still cascades the save-update operation to the locally changed association.

```
// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
Plan newPlan = new Plan( "new plan"
contract.setPlan( newPlan);
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );
newPlan = ( Contract ) session.get( Plan.class, newPlan.getId() );

// contract.getPlan() still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

tx.commit();
session.close();
```

12.2.2.2. Unidirectional one-to-many and many-to-many

Hibernate treats unidirectional one-to-many and many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks unidirectional one-to-many and many-to-many associations;

The collection can contain entities that are read-only, as well as entities that are not read-only.

Entities can be added and removed from the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in the collection if they dirty the owning entity.

12.2.3. Bidirectional associations

12.2.3.1. Bidirectional one-to-one

If a read-only entity owns a bidirectional one-to-one association:

- Hibernate does not dirty-check the association.
- updates that change the association reference to null or to refer to a different entity will not be flushed to the database.
- If automatic versioning is used, Hibernate will not increment the version due to local changes to the association.

##

If an entity is of an immutable class, and it owns a bidirectional one-to-one association, then its reference must be assigned when that entity is first created. Because the entity is automatically made read-only, these references cannot be updated.

When the owner is not read-only, Hibernate treats an association with a read-only entity the same as when the association is with an entity that is not read-only.

12.2.3.2. Bidirectional one-to-many/many-to-one

A read-only entity has no impact on a bidirectional one-to-many/many-to-one association if:

- the read-only entity is on the one-to-many side using an inverse collection;
- the read-only entity is on the one-to-many side using a non-inverse collection;
- the one-to-many side uses a non-inverse collection that contains the read-only entity

When the one-to-many side uses an inverse collection:

- a read-only entity can only be added to the collection when it is created;
- a read-only entity can only be removed from the collection by an orphan delete or by explicitly deleting the entity.

12.2.3.3. Bidirectional many-to-many

Hibernate treats bidirectional many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks bidirectional many-to-many associations.

The collection on either side of the association can contain entities that are read-only, as well as entities that are not read-only.

Entities are added and removed from both sides of the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in both sides of the collection if they dirty the entity owning the respective collections.

Transactions and Concurrency

Hibernate ################################ Hibernate ######################## JDBC ####### JTA ########### JDBC # ANSI #################DBMS############### ########################

Hibernate ## ###################### Session ### ###################

############################ SELECT FOR UPDATE ############################ API ################### API #####################

####################conversation################### Configuration#SessionFactory#### Session ###### Hibernate ##################

13.1. session ##### transaction

SessionFactory ## ############## Configuration ##################

Session ##unit of work###### ################## Session ######### JDBC Connection##### DataSource########## #########################

##

####################### Hibernate Session #################################### ######################### Session ## #######

13.1.1. #####Unit of work#

First, let's define a unit of work. A unit of work is a design pattern described by Martin Fowler as # [maintaining] a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems. #[PoEAA] In other words, its a series of operations we wish to carry out against the database together. Basically, it is a transaction, though fulfilling a unit of work will often span multiple physical database transactions (see ###### ##). So really we are talking about a more abstract notion of a transaction. The term "business transaction" is also sometimes used in lieu of unit of work.

session-per-operation ### # Session ## #############planned sequence########################### SQL ########## ## SQL

Hibernate ##
##
##
##

session-per-request
######## Hibernate ############################ Hibernate `Session` ##############
session
`Session`
##

################## Hibernate ###################### "current session" ############
##
#################### `ServletFilter` ################## AOP ######### proxy/
interception ####### EJB ##### EJB ##### Bean ###################################
CMT
Hibernate `Transaction` API

Your application code can access a "current session" to process the request by calling `sessionFactory.getCurrentSession()`. You will always get a `Session` scoped to the current database transaction. This has to be configured for either resource-local or JTA environments, see ###############.

################# ##### ###
##
############################## EJB ################################### EJB #
################################### *Open Session in View* ###################
Hibernate # Web ###################

13.1.2.

session-per-request ##
###################################### Web ###################################
##:

- ################## `Session` ##
  ###########
- 5######## "Save" ###
  #########

#################### ## ##################### ##############################
####

######################## `Session` ###
##
################################

##
##
##
################## Hibernate ##################:

- ######### - Hibernate ##
  ####################
- *###Detached#######* - ******* *session-per-request* ##############################
  ## *session-per-request-with-detached-objects* ##
- ################ - Hibernate # `Session` ################################## # JDBC ## *session-per-conversation* ##
  ############ `Session` ############################

session-per-request-with-detached-objects # *session-per-conversation* ###################
#####################################

13.1.3.

################ `Session` ############################ `Session` ################
##

#########

```
foo.getId().equals( bar.getId() )
```

JVM ###

```
foo==bar
```

###`Session` ################ ######### `Session` ########## ######################
JVM ############ Hibernate ##
################################## JVM ########################## ############ #
##########################

########## Hibernate ##
`Session`
########## synchronize ########### `Session` ############################## == ######
#####

```
#####  Session  ###  ==  ##############################################
##################  Set  #  put  ######################################
########################  JVM  ##########################  equals()  #
hashCode() ###########################################################################
#################################################################################
############################### Set ########################## Set ############
```

Set ######################## ############ Hibernate # Web ################### Hibernate ########## Java #######

13.1.4.

session-per-user-session # *session-per-application* ##################################### ###

- Session ################ HTTP ########### Bean # Swing ##################### Session ################################### HttpSession ### Hibernate Session ## ###### HttpSession ## ############ Session #######
- Hibernate ###################################### Session ######## ############ # Session ## ## ##
- The Session caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an OutOfMemoryException. One solution is to call clear() and evict() to manage the Session cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in *15######*. Keeping a Session open for the duration of a user session also means a higher probability of stale data.

13.2.

################

J2EE ############# ################# Web # Swing ###################### Hibernate ############################## Hiberante ### ### ## #CMT# ################# Bean ######## ###

############################ JTA ######### #CMT #### BMT# ###################### ## Transaction ####### API # Hibernate # ######## API ########### CMT ###### Bean #########################

Session ###################:

- ###########
- #############

- ##########
- #########

##
###############

13.2.1.

Hibernate ###################################### ##### DataSource ##### ######## ################ Hibernate ### ##########

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

You do not have to `flush()` the `Session` explicitly: the call to `commit()` automatically triggers the synchronization depending on the *FlushMode* for the session. A call to `close()` marks the end of a session. The main implication of `close()` is that the JDBC connection will be relinquished by the session. This Java code is portable and runs in both non-managed and JTA environments.

############## Hibernate ########### "current session" ############################## ######

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
```

```
}
```

Hibernate ############## RuntimeException ##### ############## ########## ################# Hibernate ###################################### SessionFactory

############## org.hibernate.transaction.JDBCTransactionFactory ###########2#### #### hibernate.current_session_context_class # "thread" ###########

13.2.2. JTA

################# ##### EJB ##### Bean #### ######## Hibernate ################## ############### JTA ################# EJB ############### JTA ############### JTA ####### Hibernate #############

Bean ###########BMT##### Transaction API ###### Hibernate ############## BMT ### ###

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

############## Session ############################## getCurrentSession() ###### JTA # UserTransaction API #############

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
```

```
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

CMT ########## Bean ##
#####:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

CMT/EJB ## Bean ######################
RuntimeException ## #### *BMT* #### *CMT* ###
Hibernate `Transaction` API
######

Hibernate ######################## JTA ###### #BMT## ### `org.hibernate.transaction.JTATransactionFactory` ## CMT ###### Bean #### `org.hibernate.transaction.CMTTransactionFactory` ################## #### `hibernate.transaction.manager_lookup_class` ###################### `hibernate.current_session_context_class` ################ `"jta"` ###########

`getCurrentSession()` ######### JTA ########################## `after_statement` ##
############################## JTA ############# `scroll()` ### `iterate()` #######
`ScrollableResults` ### `Iterator` ####### Hibernate
`finally` ######## `ScrollableResults.close()` ### `Hibernate.close(Iterator)` #######
####################### ########### ################### JTA # CMT #### `scroll()`
`iterate()`

13.2.3.

`Session` ### #`SQLException`#### ################################## `Session.close()`
`Session` ############### `Session` ################### ###### ##### Hibernate
############################## `finally` ####### `close()` ##### `Session` ############

`HibernateException` ## Hibernate ################################## # Hibernate ###
############ ##
####################
############ ################### # Hibernate ##`HibernateException` ############
##

Hibernate ################## SQLException # JDBCException ################## ## JDBCException ################## SQLException ## JDBCException.getCause() ############ Hibernate ## SessionFactory ######## SQLExceptionConverter #### SQLException #### JDBCException ################## SQLExceptionConverter ##### ### SQL ################################ ##### SQLExceptionConverterFactory # ### Javadoc ############## JDBCException ##############

- JDBCConnectionException - ##### JDBC ############
- SQLGrammarException - #### SQL ################
- ConstraintViolationException - ##################
- LockAcquisitionException - ######################################
- GenericJDBCException - ######################

13.2.4.

EJB ##
#JTA# ##### Hibernate ######################## Hibernate ########################
Hibernate ################ JTA ############ Hibernate # Transaction ############ #######

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

CMT Bean #### setTimeout() ##

13.3.

##
Hibernate

##
############################

13.3.1.

Hibernate ### `Session` ################
##
EJB
#########

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

`<version>` ##### `version` ############### Hibernate ##############################
`version` ####################

##
######### ############## ##
###

##
Hibernate ############### `Session`
############################

13.3.2.

`Session` ################### *session-per-conversation*
Hibernate ##
############# ##

################## `Session` ###### JDBC ##
##
##################################

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");
```

```
session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

foo ################ Session ### ## JDBC ############# ## ########### LockMode.READ #### Session.lock() ################# ## ############# ########## Session # FlushMode.MANUAL ### ## flush() ########################### ######## close() ####

######################## Session ############################# HttpSession #### ############ Session # ###### ### ################################## Session ################################

> **##**
>
> Hibernate ############### Session ################################# ###

Session ### Session ######### EJB # ########## Bean ######### HttpSession ######### Web ############ ##############

########### ###### *session-per-conversation* # #################################### ######## CurrentSessionContext ################ Hibernate Wiki ##############

13.3.3.

Session ########## ####### ## ################# Session ################################## Session.update() # #### Session.saveOrUpdate() # Session.merge()

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

######### Hibernate ###

############################ update() ###### LockMode.READ ##### lock() ######## #### ################################

13.3.4.

optimistic-lock ### false
##

##
##
<class> ###### optimistic-lock="all" ## Hibernate
################################### session-per-request-with-detached-objects ########
Session

<class> ###### optimistic-lock="dirty" ################## Hibernate

Hibernate
##1## UPDATE ## #### WHERE ##### ##
##
######## *on update* ################## <class> ###### select-before-update="true" #
SELECT

13.4.

JDBC
##
###############

Hibernate ##

LockMode ##### Hibernate ##

- LockMode.WRITE ## Hibernate ##########################
- LockMode.UPGRADE ################### SELECT ... FOR UPDATE ################
  #############
- LockMode.UPGRADE_NOWAIT ## Oracle # SELECT ... FOR UPDATE NOWAIT ############
  ################
- LockMode.READ ## Repeatable Read #### Serializable ############################
  ########################
- LockMode.NONE ################ Transaction ################################
  update() # saveOrUpdate() ##

################################

- LockMode ##### Session.load() ######
- Session.lock() ######
- Query.setLockMode() ######

UPGRADE #### UPGRADE_NOWAIT ###### Session.load() ############################## ################ SELECT ... FOR UPDATE ################### load() ########## ## Hibernate ############## lock() ########

############ READ ##### UPGRADE # UPGRADE_NOWAIT ###### Session.lock() ########## ########## #UPGRADE #### UPGRADE_NOWAIT #### SELECT ... FOR UPDATE ########

################################ Hibernate ################################### ########################

13.5.

Hibernate ######2.x## JDBC ############################ Session ################ ###################################### Hibernate 3.x ######## JDBC ############## ## ConnectionProvider ############## ############################ org.hibernate.ConnectionReleaseMode ############ ####

- ON_CLOSE - ###################### Hibernate ######### JDBC ################## ######################################
- AFTER_TRANSACTION - org.hibernate.Transaction ##################
- AFTER_STATEMENT ############## - ## ## org.hibernate.ScrollableResults ###########

################ hibernate.connection.release_mode ########################## ##########:

- auto ####### - ######## org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode() ######## ########################## JTATransactionFactory ## ConnectionReleaseMode.AFTER_STATEMENT #### JDBCTransactionFactory ## ConnectionReleaseMode.AFTER_TRANSACTION ################################## ##
- on_close - ConnectionReleaseMode.ON_CLOSE ################################## ########
- after_transaction - ConnectionReleaseMode.AFTER_TRANSACTION ########## JTA ## ################ ConnectionReleaseMode.AFTER_TRANSACTION ############### ######### AFTER_STATEMENT ################################
- after_statement - ConnectionReleaseMode.AFTER_STATEMENT ############### ConnectionProvider ###### (supportsAggressiveRelease()) ################### ################ ConnectionReleaseMode.AFTER_TRANSACTION ############### ############### ConnectionProvider.getConnection() ############ JDBC ###### ##

############

######### Hibernate ## Hibernate ######################

14.1.

Interceptor ## ## Interceptor # Auditable ########### createTimestamp ##### Auditable ########### lastUpdateTimestamp ############

Interceptor ######### ######### EmptyInterceptor ###########

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
```

```
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
         System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

################## Session ######## SessionFactory ##########

Session ## Interceptor ###### SessionFactory.openSession() ######################

```
Session session = sf.openSession( new AuditInterceptor() );
```

SessionFactory ############## Configuration ################## SessionFactory ## ############################## SessionFactory ################################## ### SessionFactory ########################

################ ###### ##
####

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

14.2.

############################ Hibernate3 # #### ##################################
#####################################

Session ##################1################## LoadEvent # FlushEvent
XML ####### DTD # org.hibernate.event ##############
###############1########## Hibernate # Session ###################################
##
LoadEvent ###### LoadEventListener ################################ Session
########## load() ##################

###
##

####
Hibernate ## final ########### ####
Configuration ######################## Hibernate # XML ############
############################## ######################:

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

################### Hibernate ##########

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration
```

```
>
```

##########################

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

<listener/>
###

###
############################ on/off ################

14.3. Hibernate

Hibernate ### Hiberenate3 # JACC
JAAS

###################### JAAS ####################

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

############################## <listener type="..." class="..."/> # <event type="..."><listener class="..."/></event> ####################

hibernate.cfg.xml

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

########### JACC #################

#####

Hibernate ###########100,000#######################

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

###50,000######### `OutOfMemoryException` ########## Hibernate ##################### ######## `Customer` ###################

JDBC ################################### ############### JDBC ##################10##50###########

```
hibernate.jdbc.batch_size 20
```

`identiy` #############Hibernate # JDBC ###################################

##

```
hibernate.cache.use_second_level_cache false
```

################################## `CacheMode` ###################################### ##

15.1.

`flush()` ## `clear()`

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}
```

```
tx.commit();
session.close();
```

15.2.

##
scroll()

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

15.3. StatelessSession

########## Hibernate ######## API ## ########### StatelessSession ### ## write-behind ## ### Hibernate ############# ## ######## JDBC ############

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}
```

```
tx.commit();
session.close();
```

################ Customer ########## ######### ################################## ##############

StatelessSession ################ insert(), update() # delete() ################ ##################### SQL # INSERT, UPDATE ### DELETE ######################### Session ################ save(), saveOrUpdate() # delete() ################

15.4. DML

As already discussed, automatic and transparent object/relational mapping is concerned with the management of the object state. The object state is available in memory. This means that manipulating data directly in the database (using the SQL Data Manipulation Language (DML) the statements: INSERT, UPDATE, DELETE) will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style DML statement execution that is performed through the Hibernate Query Language (*HQL*).

UPDATE # DELETE ######## (UPDATE | DELETE) FROM? ####### (WHERE ###)? ###### ##############

Some points to note:

- from ###### FROM ###############
- from ##
- No *joins*, either implicit or explicit, can be specified in a bulk HQL query. Sub-queries can be used in the where-clause, where the subqueries themselves may contain joins.
- where ##########

HQL # UPDATE ######## Query.executeUpdate() ############################ JDBC PreparedStatement.executeUpdate()

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

In keeping with the EJB3 specification, HQL UPDATE statements, by default, do not effect the *version* or the *timestamp* property values for the affected entities. However, you can force

Hibernate to reset the `version` or `timestamp` property values through the use of a `versioned update`. This is achieved by adding the `VERSIONED` keyword after the `UPDATE` keyword.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

###########`org.hibernate.usertype.UserVersionType`## `update versioned` ##########
###########

HQL # `DELETE` ########## `Query.executeUpdate()` ##############

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

`Query.executeUpdate()` ####### `int` ##
######################### HQL ################ SQL ################### joined-subclass ## joined-subclass ################
joined-subclass
######

`INSERT` ######## `INSERT INTO` ############### `select` # ###################

- INSERT INTO ... SELECT ... ################# INSERT INTO ... VALUES ... #########
  #######

  ########## SQL # `INSERT` ##### ###### ##
  ##
  ## `INSERT` ######################
- select ######## insert ################### select ##### HQL select ##############
  ## *equal* ##### Hibernate # `Type` ##
  equivalent ########################### `org.hibernate.type.DateType` ##########
  #### `org.hibernate.type.TimestampType` ####################################
  ######################################

- id ########## insert ############################## id ########### ########## select ## (##################)# ########## ########## id ## ############################## org.hibernate.id.SequenceGenerator ########## ## org.hibernate.id.PostInsertIdentifierGenerator ########################## ## org.hibernate.id.TableHiLoGenerator ## ####
- version # timestamp ##################### insert ################# ################################## select ################################## org.hibernate.type.VersionType ###### #### ###############

HQL # INSERT ########

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer
 c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

HQL: Hibernate

Hibernate # SQL ######## (#######) ###################### SQL ################## HQL ##

16.1.

Java ################################## `SeLeCT` # `sELEct` ####### `SELECT` ###### `org.hibernate.eg.FOO` # `org.hibernate.eg.Foo` ####### `foo.barSet` # `foo.BARSET` ######

############## HQL ## Java ##########################

16.2. from

######## Hibernate ##########:

```
from eg.Cat
```

This returns all instances of the class `eg.Cat`. You do not usually need to qualify the class name, since `auto-import` is the default. For example:

```
from Cat
```

In order to refer to the `Cat` in other parts of the query, you will need to assign an *alias*. For example:

```
from Cat as cat
```

######## `Cat` ######## `cat` ## `as` ######## ########################

```
from Cat cat
```

#####################################

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

######## Java ## (### `domesticCat`)#

16.3.

############################ ## ####################

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

################ ANSI SQL ######

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (############)

`inner join`# `left outer join`# `right outer join` #################

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

HQL # `with` ############################

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight
> 10.0
```

A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See ######## for more information.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

###################### where # (##########) ##################################### ## ########:

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

fetch ### iterate() ############################## #### scroll() ################ ################## fetch # setMaxResults() # setFirstResult() ################ ## eager ## fetch ######## with ## ######################## bag ### ### ############ # ############ ###################

fetch all properties ###### Hibernate

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

16.4.

HQL ################### ### # ####

######################### #### #### from ###### join ############################

join ################################### ### ######### HQL ######## ### ######## SQL

```
from Cat as cat where cat.mate.name like '%s%'
```

16.5.

######################2###########:

- ######## (###) `id` ## *id* ################################## ################### #########
- ##

id ## ################################ `id`

####

##: ######### 3.2.2 ###################### `id` ############## ## ###### ################## `id` ################## Hibernate ################

16.6. Select

`select` ##:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

######### `Cat` # `mate` ##############################:

```
select cat.mate from Cat cat
```

##:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

############### (###) ###### `Object[]` ############

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

`List`

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

Family

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

select ## as ##################

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

select new map ################

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

########## select #### Map ######

16.7.

HQL ########################

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

######################

- avg(...), sum(...), min(...), max(...)
- count(*)
- count(...), count(distinct ...), count(all...)

select ################## SQL ############:

```
select cat.weight + sum(kitten.weight)
```

```
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

SQL ######## `distinct` # `all` ################

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

16.8.

#########

```
from Cat as cat
```

`Cat` ############# `DomesticCat` ################# Hibernate #### ### Java ########## `from` ## #############:

```
from java.lang.Object o
```

`Named` ##############################:

```
from Named n, Named m where n.name = m.name
```

###2#######2#### SQL `SELECT` ######################### `order by` ################## ############ (######## `Query.scroll()` ################################)#

16.9. where

`where` ##

```
from Cat where name='Fritz'
```

######################:

```
from Cat as cat where cat.name='Fritz'
```

'Fritz' ### Cat

The following query:

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

HQL ## Foo # startDate ######### date ######### bar ################# Foo ###### ################# where

```
from Cat cat where cat.mate.name is not null
```

###################### SQL ############################

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

###################4############### SQL ############

= ##

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

The special property (lowercase) id can be used to reference the unique identifier of an object. See ############## for more information.

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

2##############################

########################## Person # country # medicareNumber ######################
######################### ############## ########

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

########2########################

See ############## for more information regarding referencing identifier properties)

class ##################################### discriminator ########## where ######## Java ######## discriminator

```
from Cat cat where cat.class = DomesticCat
```

You can also use components or composite user types, or properties of said component types. See ######### for more information.

"any" ############## id # class ############################ (AuditLog.item # <any> # ################)#

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

log.item.class # payment.class ##

16.10. Expressions

Expressions used in the where clause include the following:

- ######+, -, *, /
- 2#######=, >=, <=, <>, !=, like
- ######and, or, not
- #############( )
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of
- "####"# case case ... when ... then ... else ... end# "###"# case case when ... then ... else ... end

- ######## `...||...` ### `concat(...,...)`
- `current_date()`, `current_time()`, `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, `year(...)`,
- EJB-QL 3.0 ###############: `substring()`, `trim()`, `lower()`, `upper()`, `length()`, `locate()`, `abs()`, `sqrt()`, `bit_length()`, `mod()`
- `coalesce()` # `nullif()`
- ######## String ######## `str()`
- 2###### Hibernate ####### `cast(... as ...)` # `extract(... from ...)`############## ANSI `cast()` # `extract()` ###############
- ############################## HQL # `index()` ###
- ############### HQL ### `size()`, `minelement()`, `maxelement()`, `minindex()`, `maxindex()` # `some`, `all`, `exists`, `any`, `in` ################## `elements()` # `indices` ###########
- `sign()`# `trunc()`# `rtrim()`# `sin()` ################## SQL ######
- JDBC ############ `?`
- #########: `:name`, `:start_date`, `:x1`
- SQL ##### `'foo'`# `69`# `6.66E+2`# `'1970-01-01 10:00:01.0'`
- Java # `public static final` ### `eg.Color.TABBY`

`in` # `between` ##############:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

####################

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

`is null` # `is not null` # null

Hibernate ####### HQL query substitutions ####### boolean #################

```
<property name="hibernate.query.substitutions"
>true 1, false 0</property
>
```

########## HQL # SQL ######## `true` # `false` ###### `1` # `0` #########:

```
from Cat cat where cat.alive = true
```

######## size######### size() ########################:

```
from Cat cat where cat.kittens.size
> 0
```

```
from Cat cat where size(cat.kittens)
> 0
```

################## minindex # maxindex ################################## minelement # maxelement ##################################

```
from Calendar cal where maxelement(cal.holidays)
> current_date
```

```
from Order order where maxindex(order.items)
> 100
```

```
from Order order where minelement(order.items)
> 10000
```

######################elements # indices ################################ SQL ## any, some, all, exists, in ##########

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3
```

```
> all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

`size`# `elements`# `indices`# `minindex`# `maxindex`# `minelement`# `maxelement` # Hibernate3 # where ########################

##################arrays, lists, maps#####################where#######

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

`[]` ##################

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

######################### HQL ###### `index()` ############

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

########################### SQL ##########

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

######################### SQL #########################:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

###: ##############

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

16.11. order by

list

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

asc # desc

16.12. group by

##:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
```

```
from Foo foo join foo.names name
group by foo.id
```

having #######

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

########################## having # order by ## SQL ################ MySQL ##########

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight)
> 100
order by count(kitten) asc, sum(kitten.weight) desc
```

group by ## order by ################################ Hibernate ################## ########################## cat ################## group by cat ################## ############################

16.13.

######################## Hibernate ### SQL ############################## (##########################) #########

```
from Cat as fatcat
where fatcat.weight
> (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
```

```
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

HQL ######## select ### where ####################

Note that subqueries can also utilize `row value constructor` syntax. See ############ for more information.

16.14. HQL

Hibernate ############################## Hibernate ############################## ##

ID ## ################## SQL #### `ORDER`# `ORDER_LINE`# `PRODUCT`# `CATALOG` ### `PRICE` ####### 4######## (#####)

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate
>= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount)
> :minAmount
order by sum(price.amount) desc
```

##:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount)
> :minAmount
order by sum(price.amount) desc
```

AWAITING_APPROVAL ### ###############2######## PAYMENT, PAYMENT_STATUS ### PAYMENT_STATUS_CHANGE ######## ############ SQL

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <
> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <
> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

set ##### list ### statusChanges

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <
> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <
> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

MS SQL Server # isNull() ###### ###########3#######1########## ACCOUNT# PAYMENT# PAYMENT_STATUS# ACCOUNT_TYPE# ORGANIZATION ### ORG_USER ################## SQL

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

################## (########) ########################

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

16.15. ### UPDATE # DELETE

HQL now supports `update`, `delete` and `insert ... select ...` statements. See *#DML ### #####* for more information.

16.16. Tips & Tricks

################################:

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue()
```

#####################################:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

################################## where ###################:

```
from User usr where size(usr.messages)
```

```
>= 1
```

##:

```
select usr.id, usr.name
from User usr
    join usr.messages msg
group by usr.id, usr.name
having count(msg)
>= 1
```

################## message ####### User ########################:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

JavaBean ###################################

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

############## Query ################################:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

##:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

###################################:

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue();
```

16.17.

HQL ## `select` #########:

```
select p.name from Person p
```

```
select p.name.first from Person p
```

############################## `where` #########:

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

######## `order by` #########:

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

Another common use of components is in *row value constructors*.

16.18.

############## ANSI SQL `row value constructor` ## (`tuple` ##############) ######### ###### HQL ## #### Person ##############:

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

############################## `row value constructor` ###############:

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

`select`

```
select p.name from Person p
```

row value constructor ##########:

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

##########

Criteria

Hibernate ############ criteria ### API ##########

17.1. `Criteria`

`org.hibernate.Criteria` ################################ `Session` # `Criteria` ##### ##########

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

17.2.

`org.hibernate.criterion.Criterion` ## #### `org.hibernate.criterion.Restrictions` ############ `Criterion` ############### ############

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Restriction ####################

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Criterion ##`Restrictions` ####### ###################### SQL

```
List cats = sess.createCriteria(Cat.class)
          .add(   Restrictions.sqlRestriction("lower({alias}.name)   like   lower(?)",   "Fritz
%", Hibernate.STRING) )
    .list();
```

{alias} ##

criterion ######### Property ################## Property.forName() ####### Property # ##############

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

17.3.

org.hibernate.criterion.Order ####################

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

17.4.

By navigating associations using createCriteria() you can specify constraints upon related entities:

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

2### `createCriteria()` ## `kittens` ################## `Criteria` ######################

####################

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

#`createAlias()` #### `Criteria` ###############

##2############## `Cat` ################ kittens ######## criteria ############### #### ## ############ criteria ##### kitten ######### `ResultTransformer` #############

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

Additionally you may manipulate the result set using a left outer join:

```
List cats = session.createCriteria( Cat.class )
    .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name",
 "good%") )
       .addOrder(Order.asc("mt.age"))
       .list();
```

This will return all of the `Cats` with a mate whose name starts with "good" ordered by their mate's age, and all cats who do not have a mate. This is useful when there is a need to order or limit

in the database prior to returning complex/large result sets, and removes many instances where multiple queries would have to be performed and the results unioned by java in memory.

Without this feature, first all of the cats without a mate would need to be loaded in one query.

A second query would need to retreive the cats with mates who's name started with "good" sorted by the mates age.

Thirdly, in memory; the lists would need to be joined manually.

17.5.

`setFetchMode()` ##########################

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join. See ######## for more information.

17.6.

`org.hibernate.criterion.Example` ######################################

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

################################## null ################

`Example`

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

######### criteria ######### example ###########

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

17.7.

`org.hibernate.criterion.Projections` #### `Projection` ########## `setProjection()` ##########

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

######## criteria #####group by#################### Projection ## ####### ######## SQL # `group by` ########

An alias can be assigned to a projection so that the projected value can be referred to in restrictions or orderings. Here are two different ways to do this:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

`alias()` # `as()` ###### Projection ########### `Projection` ########################## ##############################:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

`Property.forName()` #######:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

17.8.

`DetachedCriteria` ################################### `Session` ############

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
```

```
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

DetachedCriteria ############################## Criterion ######## Subqueries #### Property ########

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight").gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

#####################:

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight").gt(avgWeightForSex) )
    .list();
```

17.9.

criteria ### criteria API #############

<natural-id>

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
```

```
</class
>
```

##

Restrictions.naturalId()

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

SQL

############ SQL ############################## Oracle # `CONNECT` ################ ################## SQL/JDBC ################## Hibernate ##############

Hibernate3 ########################### SQL ########################

18.1. Using a `SQLQuery`

SQL ####### `SQLQuery` ################## `SQLQuery` ######### `Session.createSQLQuery()` ############## API

18.1.1.

SQL

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

######### CATS ################## Object ###Object[]############################## ####### Hibernate # ResultSetMetadata #######

`ResultSetMetadata` ### `addScalar()` ######

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

########################:

- SQL ######
- #########

Object ######### `ResultSetMetdata` ################################ ID#NAME#BIRTHDATE ######## Long#String#Short ################################ ############### *

####################

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
```

```
 .addScalar("BIRTHDATE")
```

################## NAME # BIRTHDATE ######### `ResultSetMetaData` ########## ID # ###############

ResultSetMetaData ###### java.sql.Types # Hibernate ### ########## Dialect ########## ################################## Dialect # `registerHibernateType` ########## #####

18.1.2.

`addEntity()` ###### ### SQL

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

########################:

- SQL ######
- ############## SQL #######

Cat # ID # NAME # BIRTHDATE ## Cat ####### ###########

################ ### ## ##############column not found(###########)########## * ########################## ########## `Dog` # ### ######################

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

cat.getDog()

18.1.3.

########################## `Dog` ############### `addJoin()` ######################## #########

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d
 WHERE c.DOG_ID = d.D_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

############ Cat ############################ dog ################################ #########cat### Cat ##### Dog ####### #############

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE
 c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

####### Hibernate ############ SQL ## ##

18.1.4.

SQL

#####################################column alias injection#######

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = m.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

#################### Cat ## ## "c.ID"#"c.NAME" ######################### #"ID" # "NAME"#################

#########################

```
sess.createSQLQuery("SELECT {cat.*}, {m.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = m.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

#########################:

- SQL ###### #Hibernate ###########################
- ##################

########## {cat.*} # {mother.*} ## ##### Hibernate ############ SQL ### ################################# cat_log ## ################# Cat ################## ## where #####################

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
```

```
        "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
        "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

18.1.4.1.

Hibernate

###

#18.1 #########

##	##	#
#########	{[aliasname].[propertyname]	A_NAME as {item.name}
#######	{[aliasname].[componentname].[propertyname]}	CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}
################	{[aliasname].class}	DISC as {item.class}
##############	{[aliasname].*}	{item.*}
##########	{[aliasname].key}	ORGID as {coll.key}
####### ID	{[aliasname].id}	EMPID as {coll.id}
##########	{[aliasname].element}	XID as {coll.element}
################	{[aliasname].element.[propertyname]}	NAME as {coll.element.name}
#################	{[aliasname].element.*}	{coll.element.*}
All properties of the collection	{[aliasname].*}	{coll.*}

18.1.5.

SQL #### ResultTransformer

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

########################:

- SQL ######
- #########

################## NAME # BIRTHDATE ########################## `CatDTO` ########
##

18.1.6.

################################## SQL ######################################
########

18.1.7.

SQL ######################:name####################:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

18.2. #### SQL

Named SQL queries can also be defined in the mapping document and called in exactly the same way as a named HQL query (see ##############). In this case, you do *not* need to call `addEntity()`.

#18.1 Named sql query using the <sql-query> maping element

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

#18.2 Execution of a named query

```
List people = sess.getNamedQuery("persons")
```

```
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

The <return-join> element is use to join associations and the <load-collection> element is used to define queries which initialize collections,

#18.3 Named sql query with association

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

SQL ################## <return-scalar> ############ Hibernate ########### ###:

#18.4 Named query returning a scalar

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

################ <resultset> ################################ setResultSetMapping() API ###############

#18.5 <resultset> mapping used to externalize mapping information

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>
```

```
<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

hbm ######################### Java

#18.6 Programmatically specifying the result mapping information

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

So far we have only looked at externalizing SQL queries using Hibernate mapping files. The same concept is also available with anntations and is called named native queries. You can use `@NamedNativeQuery` (`@NamedNativeQueries`) in conjunction with `@SqlResultSetMapping` (`@SqlResultSetMappings`). Like `@NamedQuery`, `@NamedNativeQuery` and `@SqlResultSetMapping` can be defined at class level, but their scope is global to the application. Lets look at a view examples.

#18.7#Named SQL query using @NamedNativeQuery together with @SqlResultSetMapping# shows how a `resultSetMapping` parameter is defined in `@NamedNativeQuery`. It represents the name of a defined `@SqlResultSetMapping`. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property. In the example 2 entities, `Night` and `Area`, are returned and each property is declared and associated to a column name, actually the column name retrieved by the query.

In *#18.8#Implicit result set mapping#* the result set mapping is implicit. We only describe the entity class of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column.

Finally, if the association to a related entity involve a composite primary key, a `@FieldResult` element should be used for each foreign key column. The `@FieldResult` name is composed of the property name for the relationship, followed by a dot ("."), followed by the name or the field

or property of the primary key. This can be seen in *#18.9#Using dot notation in @FieldResult for specifying associations #*.

#18.7 Named SQL query using `@NamedNativeQuery` together with `@SqlResultSetMapping`

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
    + " night.night_date, area.id aid, night.area_id, area.name "
    + "from Night night, Area area where night.area_id = area.id",
                resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)
```

#18.8 Implicit result set mapping

```
@Entity
@SqlResultSetMapping(name="implicit",
                    entities=@EntityResult(entityClass=SpaceShip.class))
@NamedNativeQuery(name="implicitSample",
                query="select * from SpaceShip",
                resultSetMapping="implicit")
public class SpaceShip {
    private String name;
    private String model;
    private double speed;

    @Id
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Column(name="model_txt")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
```

```
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }
}
```

#18.9 Using dot notation in @FieldResult for specifying associations

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=SpaceShip.class,
            fields = {
                    @FieldResult(name="name", column = "name"),
                    @FieldResult(name="model", column = "model"),
                    @FieldResult(name="speed", column = "speed"),
                    @FieldResult(name="captain.firstname", column = "firstn"),
                    @FieldResult(name="captain.lastname", column = "lastn"),
                    @FieldResult(name="dimensions.length", column = "length"),
                    @FieldResult(name="dimensions.width", column = "width")
                    }),
        columns = { @ColumnResult(name = "surface"),
                    @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
     query="select name, model, speed, lname as lastn, fname as firstn, length, width, length
 * width as surface from SpaceShip",
    resultSetMapping="compositekey")
} )
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManyToOne(fetch= FetchType.LAZY)
    @JoinColumns( {
            @JoinColumn(name="fname", referencedColumnName = "firstname"),
            @JoinColumn(name="lname", referencedColumnName = "lastname")
            } )
    public Captain getCaptain() {
```

```
        return captain;
    }

    public void setCaptain(Captain captain) {
        this.captain = captain;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }

    public void setDimensions(Dimensions dimensions) {
        this.dimensions = dimensions;
    }
}

@Entity
@IdClass(Identity.class)
public class Captain implements Serializable {
    private String firstname;
    private String lastname;

    @Id
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    @Id
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

If you retrieve a single entity using the default mapping, you can specify the `resultClass` attribute instead of `resultSetMapping`:

```
@NamedNativeQuery(name="implicitSample",       query="select       *       from
 SpaceShip", resultClass=SpaceShip.class)
public class SpaceShip {
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the `@SqlResultsetMapping` through `@ColumnResult`. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

#18.10 Scalar values via `@ColumnResult`

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar",     query="select     length*width     as     dimension     from
 SpaceShip", resultSetMapping="scalar")
```

An other query hint specific to native queries has been introduced: `org.hibernate.callable` which can be true or false depending on whether the query is a stored procedure or not.

18.2.1. ################## return-property

########## {} ########## `<return-property>` ######################

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` ## {} ############

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
```

```
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

############ {} ######### `<return-property>` ###################################### #######

discriminator ########## discriminator ########### `<return-discriminator>` ####

18.2.2.

Hibernate ######3## ################# Hibernate ########1################################### Oracle 9## ######################################:

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

Hibernate #######################################

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
```

```
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

<return-join> # <load-collection> ######

18.2.2.1.

Hibernate ## #### Hibernate ## session.connection() ## ## ######

setFirstResult()/setMaxResults() #####################################

################# SQL92 ######## { ? = call functionName(<parameters>) } # { ? = call procedureName(<parameters>} ##############################

Oracle ################:

- ## OUT ############ Oracle 9 # 10 ### SYS_REFCURSOR ############### Oracle ## REF CURSOR ############### Oracle #### ##########

Sybase # MS SQL ######################:

- ## Hibernate #1####### ####################################
- ########## SET NOCOUNT ON ###

18.3. ################## SQL

Hibernate3 can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or inidividual column level. This section describes statement overrides. For columns, see *#Column transformers: read and write expressions#*. *#18.11#Custom CRUD via annotations#* shows how to define custom SQL operatons using annotations.

#18.11 Custom CRUD via annotations

```
@Entity
@Table(name="CHAOS")
@SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname, id) VALUES(?,upper(?),?,?)")
@SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = upper(?), nickname = ? WHERE id = ?")
@SQLDelete( sql="DELETE CHAOS WHERE id = ?")
@SQLDeleteAll( sql="DELETE CHAOS")
@Loader(namedQuery = "chaos")
@NamedNativeQuery(name="chaos", query="select id, size, name, lower( nickname ) as nickname from
 CHAOS where id= ?", resultClass = Chaos.class)
```

```
public class Chaos {
    @Id
    private Long id;
    private Long size;
    private String name;
    private String nickname;
```

`@SQLInsert`, `@SQLUpdate`, `@SQLDelete`, `@SQLDeleteAll` respectively override the INSERT, UPDATE, DELETE, and DELETE all statement. The same can be achieved using Hibernate mapping files and the `<sql-insert>`, `<sql-update>` and `<sql-delete>` nodes. This can be seen in *#18.12#Custom CRUD XML#*.

#18.12 Custom CRUD XML

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

If you expect to call a store procedure, be sure to set the `callable` attribute to `true`. In annotations as well as in xml.

To check that the execution happens correctly, Hibernate allows you to define one of those three strategies:

- none: no check is performed: the store procedure is expected to fail upon issues
- count: use of rowcount to check that the update is successful
- param: like COUNT but using an output parameter rather that the standard mechanism

To define the result check style, use the `check` parameter which is again available in annoations as well as in xml.

You can use the exact same set of annotations respectively xml nodes to override the collection related statements -see *#18.13#Overriding SQL statements for collections using annotations#*.

#18.13 Overriding SQL statements for collections using annotations

```
@OneToMany
@JoinColumn(name="chaos_fk")
@SQLInsert( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = ? where id = ?")
@SQLDelete( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = null where id = ?")
private Set<CasimirParticle> particles = new HashSet<CasimirParticle>();
```

####

The parameter order is important and is defined by the order Hibernate handles properties. You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL through annotations or mapping files as that will override the Hibernate generated static sql)

Overriding SQL statements for secondary tables is also possible using `@org.hibernate.annotations.Table` and either (or all) attributes `sqlInsert`, `sqlUpdate`, `sqlDelete`:

#18.14 Overriding SQL statements for secondary tables

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "`Cat nbr1`"),
    @SecondaryTable(name = "Cat2"})
@org.hibernate.annotations.Tables( {
    @Table(appliesTo = "Cat", comment = "My cat table" ),
    @Table(appliesTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchMode.SELECT,
        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
} )
public class Cat implements Serializable {
```

The previous example also shows that you can give a comment to a given table (primary or secondary): This comment will be used for DDL generation.

####

The SQL is directly executed in your database, so you can use any dialect you like. This will, however, reduce the portability of your mapping if you use database specific SQL.

Last but not least, stored procedures are in most cases required to return the number of rows inserted, updated and deleted. Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations:

#18.15 Stored procedures and their return value

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN
```

```
    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

18.4. ########### SQL

You can also declare your own SQL (or HQL) queries for entity loading. As with inserts, updates, and deletes, this can be done at the individual column level as described in *#Column transformers: read and write expressions#* or at the statement level. Here is an example of a statement level override:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

##:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

#########################

###################################:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
```

```
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

##:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

The annotation equivalent `<loader>` is the @Loader annotation as seen in *#18.11#Custom CRUD via annotations#*.

###########

Hibernate3 ### *Hibernate filter* ############### ################### Hibernate ######################

19.1. Hibernate

Hibernate3 ### ## #where# ### ## ###

Using annotatons filters are defined via `@org.hibernate.annotations.FilterDef` or `@org.hibernate.annotations.FilterDefs`. A filter definition has a `name()` and an array of parameters(). A parameter will allow you to adjust the behavior of the filter at runtime. Each parameter is defined by a `@ParamDef` which has a name and a type. You can also define a `defaultCondition()` parameter for a given `@FilterDef` to set the default condition to use when none are defined in each individual `@Filter`. `@FilterDef`(s) can be defined at the class or package level.

We now need to define the SQL filter clause applied to either the entity load or the collection load. `@Filter` is used and placed either on the entity or the collection element. The connection between `@FilterName` and `@Filter` is a matching name.

#19.1 @FilterDef and @Filter annotations

```
@Entity
@FilterDef(name="minLength", parameters=@ParamDef( name="minLength", type="integer" ) )
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

When the collection use an association table as a relational representation, you might want to apply the filter condition to the association table itself or to the target entity table. To apply the constraint on the target entity, use the regular `@Filter` annotation. However, if you want to target the association table, use the `@FilterJoinTable` annotation.

#19.2 Using `@FilterJoinTable` for filterting on the association table

```
@OneToMany
    @JoinTable
    //filter on the target entity table
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
    //filter on the association table
```

```
    @FilterJoinTable(name="security", condition=":userlevel >= requredLevel")
    public Set<Forest> getForests() { ... }
```

Using Hibernate mapping files for defining filters the situtation is very similar. The filters must first be defined and then attached to the appropriate mapping elements. To define a filter, use the `<filter-def/>` element within a `<hibernate-mapping/>` element:

#19.3 Defining a filter definition via `<filter-def>`

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

This filter can then be attached to a class or collection (or, to both or multiples of each at the same time):

#19.4 Attaching a filter to a class or collection using `<filter>`

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>

    <set ...>
        <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
    </set>
</class>
```

Session ####### enableFilter(String filterName)# getEnabledFilter(String filterName)# disableFilter(String filterName) ################################ ###### # Filter ############## Session.enabledFilter() ############################ ################################

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

org.hibernate.Filter ################ Hibernate ###################################

##################################

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Department"/>
```

```
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

##:

```
Session session = ...;
session.enableFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

HQL ###100###########
#########

(HQL
##

##
############## `<filter-def/>` ####### CDATA ####################################

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

##
###

XML

XML Mapping is an experimental feature in Hibernate 3.0 and is currently under active development.

20.1. XML

Hibernate ###### POJO ########################## XML ################## XML # ### POJO ##

Hibernate # XML ########### API ### dom4j ################### dom4j ############### ## XML ################# dom4j ########## ###### Hibernate ################################## #### `persist()`, `saveOrUpdate()`, `merge()`, `delete()`, `replicate()` #### (################)#

#########/######## JMS ################# SOAP # XSLT ########################### ###

###################### XML ## ####### XML ######################

20.1.1. XML

POJO # XML

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class
>
```

20.1.2. XML

POJO

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class
>
```

########## dom4j ##########/########java # Map############################ HQL ####################

20.2. XML

Hibernate ######### node ############### XML ################################ ##### node ###############1#############

- "element-name" - #### XML ###########
- "@attribute-name" - #### XML ###########
- "." - #############
- "element-name/@attribute-name" - ##########################

######################## embed-xml ############### embed-xml="true" ########### ######## (#########) # XML ##################### XML ################## embed-xml="false" ################################### XML ############################

############## embed-xml="true" ################### XML #############

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>
```

```
    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class
>
```

######### account ######### account # id ######################## HQL ######

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

###################:

```
<customer id="123456789">
    <account short-desc="Savings"
>987632567</account>
    <account short-desc="Credit Card"
>985612323</account>
    <name>
        <first-name
>Gavin</first-name>
        <initial
>A</initial>
        <last-name
>King</last-name>
    </name>
    ...
</customer
>
```

<one-to-many> ###### embed-xml="true" ########################

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance
>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance
>-2370.34</balance>
    </account>
    <name>
        <first-name
>Gavin</first-name>
        <initial
>A</initial>
        <last-name
>King</last-name>
    </name>
    ...
</customer
>
```

20.3. XML

XML ## dom4j ####################

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
```

```
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

XML ##############/################ Hibernate # `replicate()` ########################

##########

21.1.

Hibernate ############################## O/R ##################### HQL # `Criteria`

Hibernate3 ###################:

- ###### - Hibernate # `OUTER JOIN` ########################## `SELECT` #######
- ######## - 2### `SELECT` ########################## `lazy="false"` ################ #######2### select ############################
- ########## - 2### `SELECT` ## `lazy="false"` ########################2### select ############################
- ####### - ###################### - Hibernate #########################1## `SELECT` # ###################################

Hibernate ###############:

- ###### - ##
- ############ - ##(################## ######)
- ################# - ## Hibernate ####### ##
- ######## - ########## getter ###################################
- ############ - ## ## ################################
- ######## - ## #########

##############: ## ############### ##### ########## SQL ################### `fetch` ################## `lazy` ## ####

21.1.1.

######## Hibernate3 ## ################################

#:`hibernate.default_batch_fetch_size` ########## Hibernate ######################## ################################

#################################### Hibernate # session ######################## ####################

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

`Session` ########### permissions ## *Hibernate* ############################### ## ###############

`lazy="false"` ### ### Hibernate

Hibernate3 ##

21.1.2.

################ N+1 ###:

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

############ #### ##############:

- `get()` # `load()` #####
- ########################
- `Criteria` ###
- ###### ####### HQL ###

HQL
SELECT

HQL # `left join fetch`
################################### Hibernate ###################################
`Criteria` #### API ### `setFetchMode(FetchMode.JOIN)`

`get()` # `load()` ############################## `Criteria`

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

######### ORM ######## "fetch plan" ###############

N+1 ########################2#################

21.1.3.

################ Hibernate ###
################################### Hibernate ####### CGLIB ####################
################################

######## Hibernate3 ################################### `many-to-one` # `one-to-one` #
##################

########## `proxy` ## Hibernate
################## ###
#########################

##

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

`Cat` ######## `DomesticCat` ########################## `DomesticCat` ##########:

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

######### == ###############

```
Cat cat = (Cat) session.load(Cat.class, id);            // instantiate a Cat proxy
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // acquire new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

##
########:

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

`final` #### `final` ########### CGLIB

######################### (##########################) ############################
##

####### Java ##
##

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

Then proxies for instances of `Cat` and `DomesticCat` can be returned by `load()` or `iterate()`.

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.createQuery("from CatImpl as cat where cat.name='fritz'").iterate();
Cat fritz = (Cat) iter.next();
```

Note

`list()` does not usually return proxies.

######################## `Cat` ######################## `CatImpl` ########

######### ###### #########

- `equals()` ####### `equals()` ###############

- `hashCode()` ####### `hashCode()` ##############
- #### getter ####

Hibernate # `equals()` # `hashCode()` ######################

`lazy="proxy"` ###### `lazy="no-proxy"` ######################################

21.1.4.

`LazyInitializationException` ## `Session` ###################################### Hibernate ##

`Session` ## `cat.getSex()` # `cat.getKittens().size()` ## ##############

static ##### `Hibernate.initialize()` # `Hibernate.isInitialized()` ################ ############################## `Hibernate.initialize(cat)` ## `Session` ########### `cat` ################ `Hibernate.initialize( cat.getKittens() )` # kittens ######### ############

################################### `Session` ################################### ############ Hibernate ## ######################## `Session` ###################################2########## #:

- Web ## `Session` ################### ############ *Open Session in View* ## ### `Session` ############################## #### Hibernate # Wiki ###### "Open Session in View" ###############
- ################################ Web ###################################### ##########################/ Web ## ######################## Web ################## `Hibernate.initialize()` # ###################################### Hibernate #### `FETCH` ## `Criteria` # `FetchMode.JOIN` ###################### *Session Facade* ######### *Command* ###### ############
- ################################### `merge()` # `lock()` ####### `Session` ####### ## Hibernate ########### # ########## #

###

##

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

createFilter() ##:

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

21.1.5.

Hibernate ## Hibernate ########## ### ###################

Session #######25## Cat ############## ### Cat # owner ### Person ########## Person #### lazy="true" ##################### ##### Cat ######## getOwner() ####### Hibernate ########25## SELECT #### owner #### ################ Person ####### batch-size

```
<class name="Person" batch-size="10">...</class>
```

Hibernate ############################ 10, 10, 5 ###

Person # Cat ################ 10 ## Person # Sesssion ################ Person ######## getCats() ##########10## SELECT ##### #### Person ####### cats ###################### Hibernate

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

batch-size # 3 #### Hibernate # 4 ## SELECT # 3 ## 3 ## 3 ## 1 ######################## Session ##############################

set # #######

21.1.6.

####################################### Hibernate ################################ ###

21.1.7. Fetch profiles

Another way to affect the fetching strategy for loading associated objects is through something called a fetch profile, which is a named configuration associated with the org.hibernate.SessionFactory but enabled, by name, on the org.hibernate.Session.

Once enabled on a `org.hibernate.Session`, the fetch profile will be in affect for that `org.hibernate.Session` until it is explicitly disabled.

So what does that mean? Well lets explain that by way of an example which show the different available approaches to configure a fetch profile:

#21.1 Specifying a fetch profile using `@FetchProfile`

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {

  @FetchProfile.FetchOverride(entity = Customer.class, association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   private long customerNumber;

   @OneToMany
   private Set<Order> orders;

   // standard getter/setter
   ...
}
```

#21.2 Specifying a fetch profile using `<fetch-profile>` outside `<class>` node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>
    <class name="Order">
        ...
    </class>
    <fetch-profile name="customer-with-orders">
        <fetch entity="Customer" association="orders" style="join"/>
    </fetch-profile>
</hibernate-mapping>
```

#21.3 Specifying a fetch profile using `<fetch-profile>` inside `<class>` node

```
<hibernate-mapping>
```

```
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
        <fetch-profile name="customer-with-orders">
            <fetch association="orders" style="join"/>
        </fetch-profile>
    </class>
    <class name="Order">
        ...
    </class>
</hibernate-mapping>
```

Now normally when you get a reference to a particular customer, that customer's set of orders will be lazy meaning we will not yet have loaded those orders from the database. Normally this is a good thing. Now lets say that you have a certain use case where it is more efficient to load the customer and their orders together. One way certainly is to use "dynamic fetching" strategies via an HQL or criteria queries. But another option is to use a fetch profile to achieve that. The following code will load both the customer *and*their orders:

#21.4 Activating a fetch profile for a given `Session`

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" );  // name matches from mapping
Customer customer = (Customer) session.get( Customer.class, customerId );
```

> **##**
>
> `@FetchProfile` definitions are global and it does not matter on which class you place them. You can place the `@FetchProfile` annotation either onto a class or package (package-info.java). In order to define multiple fetch profiles for the same class or package `@FetchProfiles` can be used.

Currently only join style fetch profiles are supported, but they plan is to support additional styles. See *HHH-3414* [http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414] for details.

21.1.8.

Hibernate3 ##################################### ########## ######################### ## ###

###################################### `lazy` #############:

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

Hibernate
####################

############### Ant ###########:

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

HQL # Criteria
###

HQL # `fetch all properties` ###

21.2. #2########

Hibernate # `Session` ########################## class-by-class # collection-by-collection # ########### JVM ### # `SessionFactory` ###
#####

You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`. Hibernate is bundled with a number of built-in integrations with the open-source cache providers that are listed in *#21.1############*. You can also implement your own and plug it in as outlined above. Note that versions prior to Hibernate 3.2 use EhCache as the default cache provider.

#21.1

#####	########	###	#######	######### ####
Hashtable## ######### #####	`org.hibernate.cache.HashtableCacheP##vider`			yes
EHCache	`org.hibernate.cache.EhCacheProvider`	memory, disk, transactional, clustered	yes	yes
OSCache	`org.hibernate.cache.OSCacheProvider`	########		yes
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	####ip ## ######	yes###### ###	
JBoss Cache 1.x	`org.hibernate.cache.TreeCacheProvider`	####ip ## ######### #######	yes####	yes###### ###
JBoss Cache 2	`org.hibernate.cache.jbc.JBossCacheRegionFactory`	####ip ## ######### #######	yes (replication or invalidation)	yes###### ###

21.2.1.

As we have done in previous chapters we are looking at the two different possibiltites to configure caching. First configuration via annotations and then via Hibernate mapping files.

By default, entities are not part of the second level cache and we recommend you to stick to this setting. However, you can override this by setting the `shared-cache-mode` element in your `persistence.xml` file or by using the `javax.persistence.sharedCache.mode` property in your configuration. The following values are possible:

- `ENABLE_SELECTIVE` (Default and recommended value): entities are not cached unless explicitly marked as cacheable.
- `DISABLE_SELECTIVE`: entities are cached unless explicitly marked as not cacheable.
- `ALL`: all entities are always cached even if marked as non cacheable.
- `NONE`: no entity are cached even if marked as cacheable. This option can make sense to disable second-level cache altogether.

The cache concurrency strategy used by default can be set globaly via the `hibernate.cache.default_cache_concurrency_strategy` configuration property. The values for this property are:

- `read-only`
- `read-write`
- `nonstrict-read-write`
- `transactional`

##

It is recommended to define the cache concurrency strategy per entity rather than using a global one. Use the `@org.hibernate.annotations.Cache` annotation for that.

#21.5 Definition of cache concurrency strategy via `@Cache`

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

Hibernate also let's you cache the content of a collection or the identifiers if the collection contains other entities. Use the `@Cache` annotation on the collection property.

#21.6 Caching collections using annotations

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

*#21.7#@Cache annotation with attributes#*shows the `@org.hibernate.annotations.Cache` annotations with its attributes. It allows you to define the caching strategy and region of a given second level cache.

#21.7 `@Cache` annotation with attributes

```
@Cache(
    CacheConcurrencyStrategy usage();          (1)
    String region() default "";                (2)
    String include() default "all";            (3)
)
```

1. usage: the given cache concurrency strategy (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)
2. region (optional): the cache region (default to the fqcn of the class or the fq role name of the collection)
3. `include` (optional): all to include all properties, non-lazy to only include non lazy properties (default all).

Let's now take a look at Hibernate mapping files. There the `<cache>` element of a class or collection mapping is used to configure the second level cache. Looking at *#21.8#The Hibernate <cache> mapping element#* the parallels to anotations is obvious.

#21.8 The Hibernate `<cache>` mapping element

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"    (1)
    region="RegionName"                                                (2)
    include="all|non-lazy"                                             (3)
/>
```

1. `usage` (##) ########## `transactional`# `read-write`# `nonstrict-read-write` ## # `read-only`
2. `region` (################################) 2####################
3. `include` (###### `all` #########) `non-lazy` ## ###### lazy ########## `lazy="true"` ##

Alternatively to `<cache>`, you can use `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

Let's now have a closer look at the different usage strategies

21.2.2. read only

`read-only` ##########################

21.2.3. read/write

######################## `read-write` ## ### JTA ########## JTA `TransactionManager` ############ `hibernate.transaction.manager_lookup_class` ################### #### `Session.close()` # `Session.disconnect()` ################################## ## ########## #

21.2.4. ###### read/write

nonstrict-read-write ########################## JTA ########## hibernate.transaction.manager_lookup_class ########################## Session.close() # Session.disconnect()

21.2.5. transactional

transactional ######## JBoss TreeCache ## ######### JTA ########### hibernate.transaction.manager_lookup_class ########## ####

21.2.6. Cache-provider/concurrency-strategy compatibility

####

##

####################################

#21.2

#####	read-only	nonstrict-read-write	read-write	transactional
Hashtable##### ##########	yes	yes	yes	
EHCache	yes	yes	yes	yes
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss Cache 1.x	yes			yes
JBoss Cache 2	yes			yes

21.3.

####### save() # update() # saveOrUpdate() ######### load() # get() # list() # iterate() # scroll() ################################ Session ################

flush() ## ########################## evict()

#21.9 Explcitly evicting a cached instance from the first level cache using `Session.evict()`

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

`Session` ## `contains()` ############

`Session.clear()`

############ `SessionFactory` ##
######################

#21.10 Second-level cache eviction via `SessionFactoty.evict()` and `SessionFacyory.evictCollection()`

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

`CacheMode` ##

- `CacheMode.NORMAL` - ############################
- `CacheMode.GET` - ##
- `CacheMode.PUT` - ##
- `CacheMode.REFRESH` - ## `hibernate.cache.use_minimal_puts` ## ##########

############################## `Statistics` API ########:

#21.11 Browsing the second-level cache entries via the `Statistics` API

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

Hibernate ######:

#21.12 Enabling Hibernate statistics

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

21.4.

Query result sets can also be cached. This is only useful for queries that are run frequently with the same parameters.

21.4.1. Enabling query caching

Caching of query results introduces some overhead in terms of your applications normal transactional processing. For example, if you cache results of a query against Person Hibernate will need to keep track of when those results should be invalidated because changes have been committed against Person. That, coupled with the fact that most applications simply gain no benefit from caching query results, leads Hibernate to disable caching of query results by default. To use query caching, you will first need to enable the query cache:

```
hibernate.cache.use_query_cache true
```

This setting creates two new cache regions:

- `org.hibernate.cache.StandardQueryCache`, holding the cached query results
- `org.hibernate.cache.UpdateTimestampsCache`, holding timestamps of the most recent updates to queryable tables. These are used to validate the results as they are served from the query cache.

####

If you configure your underlying cache implementation to use expiry or timeouts is very important that the cache timeout of the underlying cache region for the UpdateTimestampsCache be set to a higher value than the timeouts of any of the query caches. In fact, we recommend that the the UpdateTimestampsCache region not be configured for expiry at all. Note, in particular, that an LRU cache expiry policy is never appropriate.

As mentioned above, most queries do not benefit from caching or their results. So by default, individual queries are not cached even after enabling query caching. To enable results caching for a particular query, call `org.hibernate.Query.setCacheable(true)`. This call allows the query to look for existing cache results or add its results to the cache when it is executed.

##

The query cache does not cache the state of the actual entities in the cache; it caches only identifier values and results of value type. For this reaso, the query cache should always be used in conjunction with the second-level cache for those entities expected to be cached as part of a query result cache (just as with collection caching).

21.4.2. Query cache regions

############################## `Query.setCacheRegion()` ############################## ##########

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
        .setEntity("blogger", blogger)
        .setMaxResults(15)
        .setCacheable(true)
        .setCacheRegion("frontpages")
        .list();
```

If you want to force the query cache to refresh one of its regions (disregard any cached results it finds there) you can use `org.hibernate.Query.setCacheMode(CacheMode.REFRESH)`. In conjunction with the region you have defined for the given query, Hibernate will selectively force the results cached in that particular region to be refreshed. This is particularly useful in cases where underlying data may have been updated via a separate process and is a far more efficient alternative to bulk eviction of the region via `org.hibernate.SessionFactory.evictQueries()`.

21.5.

In the previous sections we have covered collections and their applications. In this section we explore some more issues in relation to collections at runtime.

21.5.1.

Hibernate #3#########################:

- ########
- ######
- ######

Hibernate ##

- ###############
- set
- bag

################## ############ # <key> # <index> ############################## ############################## Hibernate ######################################

set # <key> ## ## ############################## SchemaExport #### <set> ################## not-null="true" ###############

<idbag> ##

bag ########## bag ## Hibernate ## ################## Hibernate ################################### DELETE ######## ###################################

####Hibernate###################################

21.5.2. ################## list#map#idbag#set

############################## set ###################################

set ################### Set ######## ## Hibernate ################### UPDATE ##### Set ############## INSERT # DELETE ###

##############################list#map#idbag ################inverse ############ ####### set ############## Hibernate ############ set ########################## "set" ######################

############# Hibernate ############################## inverse="true" ########## ##

21.5.3. inverse ############## bag # list

bag ########## bag #### list ### set ############################## inverse="true" ########################## bag ################## bag # list ############## Collection.add() # Collection.addAll() # bag # List #### true ############### # Set ###################################

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);  //no need to fetch the collection!
```

```
sess.flush();
```

21.5.4.

Hibernate ######################## list.clear() ## Hibernate # DELETE #########

###20## Hibernate #### INSERT ##### DELETE ### #### ######## bag ######################

####18#########2#########3################################

- 18###########3######
- ############ DELETE # SQL #########5###################

Hibernate ######2################################# Hibernate ##################### ###

##2## ###################################

########## inverse="true" ###################

21.6.

####################################### Hibernate ###################### Hibernate ###### SessionFactory ##########

21.6.1. SessionFactory

SessionFactory ###############2################# sessionFactory.getStatistics() ######### Statistics ###############

StatisticsService MBean ########## Hibernate # JMX #######################1## MBean ##### SessionFactory ########### SessionFactory ###### MBean ############## ########################:

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

SessionFactory ############################

- ###### hibernate.generate_statistics # false ####
- ##### sf.getStatistics().setStatisticsEnabled(true) ### hibernateStatsBean.setStatisticsEnabled(true) #######

clear() ############################## logSummary() ######## logger ########## #info

21.6.2.

######################### Statistics ######### API ########3#########:

- ########## Session ############################ JDBC #####################
- ###
- ##

Java #### ############### Hibernate # JVM ######################10####################

getter ### ## HQL # SQL ################# ############## Statistics # EntityStatistics # CollectionStatistics # SecondLevelCacheStatistics # QueryStatistics API # javadoc #######################:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
```

```
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

getQueries() # getEntityNames()# getCollectionRoleNames() # getSecondLevelCacheRegionNames() ##################

#########

Hibernate ######################## Eclipse ######################## Ant ###############

Hibernate Tools ############################ Ant ######## Eclipse IDE ############:

- *#########:* Hibernate # XML ##/ ########################## XML ############
- *Console:* ## ################### HQL ############################
- ######## Hibernate # Eclipse ################################ Hibernate ######## (cfg.xml) ########################## POJO ######### Hibernate ################### ##
-

######## *Hibernate Tools* ########################

Hibernate ########## *SchemaExport* ### `hbm2ddl` #####(Hibernate ################ #)#

22.1.

DDL # Hibernate ## ########## (########) ##

DDL ######################## `hibernate.dialect` ###### SQL # ## ### ########## #

##

22.1.1.

Hibernate ################## `length`# `precision`# `scale` ##########################

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

`not-null` ############ `NOT NULL` ######### `unique` ############ `UNIQUE` ############ ##########

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

unique-key ## unique-key ################## ### ##################################

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

index ##

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

foreign-key ##

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

############## <column> ###################################:

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property
>
```

default ##################### (## ###)#

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property
>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property
>
```

sql-type ########## Hibernate ### SQL ##########################

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property
>
```

check ######################

```
<property name="foo" type="integer">
    <column name="foo" check="foo
> 10"/>
</property
>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class
>
```

The following table summarizes these optional attributes.

#22.1

##	#	##
length	##	######
precision	##	#### DECIMAL #####precision#
scale	##	#### DECIMAL #######scale#
not-null	true|false	#### null ##############
unique	true|false	######################
index	index_name	(######)###############
unique-key	unique_key_name	######################
foreign-key	foreign_key_name	<one-to-one># <many-to-one># <key># ## <many-to-many> ###################### ###################### inverse="true" ## SchemaExport ######################
sql-type	SQL column type	###################### (<column> ########)
default	SQL #	################
check	SQL #	######### SQL ##############

<comment> ##############################

```
<class name="Customer" table="CurCust">
    <comment
>Current customers only</comment>
    ...
</class
>
```

```
<property name="balance">
    <column name="bal">
        <comment
>Balance in USD</comment>
    </column>
</property
>
```

########## DDL # `comment on table` # `comment on column` ########

22.1.2.

`SchemaExport` ######### DDL ########### DDL #############

The following table displays the `SchemaExport` command line options

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

#22.2 `SchemaExport`

#####	##
`--quiet`	##################
`--drop`	###############
`--create`	###############
`--text`	##################
`--output=my_schema.ddl`	DDL ##################
`--naming=eg.MyNamingStrategy`	`NamingStrategy` ###
`--config=hibernate.cfg.xml`	XML ###### Hibernate ############
`--properties=hibernate.properties`	##########################
`--format`	########### SQL ###################
`--delimiter=;`	##################

######### `SchemaExport` ##############

```
Configuration cfg = ....;
```

```
new SchemaExport(cfg).create(false, true);
```

22.1.3.

################################

- `-D`*<property>* ################
- `hibernate.properties` ######
- `--properties` ##################

################

#22.3 SchemaExport

######	##
`hibernate.connection.driver_class`	jdbc #########
`hibernate.connection.url`	jdbc # url
`hibernate.connection.username`	###########
`hibernate.connection.password`	#########
`hibernate.dialect`	########

22.1.4. Ant

Ant ########### `SchemaExport` #############:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target
>
```

22.1.5.

`SchemaUpdate` ############################ `SchemaUpdate` # JDBC ###### API ####### ########### JDBC ##############################

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

#22.4 `SchemaUpdate`

#####	##
`--quiet`	#################
`--text`	#######################
`--naming=eg.MyNamingStrategy`	`NamingStrategy` ###
`--properties=hibernate.properties`	#########################
`--config=hibernate.cfg.xml`	`.cfg.xml` #######

######### `SchemaUpdate` ##############

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

22.1.6. ##################### Ant

Ant ####### `SchemaUpdate` ##############

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target
>
```

22.1.7. Schema validation

`SchemaValidator` ##
`SchemaValidator` # JDBC ###### API ############################## JDBC ###########
##############################

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

The following table displays the `SchemaValidator` command line options:

#22.5 `SchemaValidator`

#####	##
`--naming=eg.MyNamingStrategy`	`NamingStrategy` ###
`--properties=hibernate.properties`	##########################
`--config=hibernate.cfg.xml`	`.cfg.xml` #######

`SchemaValidator` ######################:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

22.1.8. ############## Ant

Ant ####### `SchemaValidator` #######:

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemavalidator>
</target
>
```

Additional modules

Hibernate Core also offers integration with some external modules/projects. This includes Hibernate Validator the reference implementation of Bean Validation (JSR 303) and Hibernate Search.

23.1. Bean Validation

Bean Validation standardizes how to define and declare domain model level constraints. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. Bean Validation can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Bean Validation and its reference implementation Hibernate Validator has been designed for that purpose.

The integration between Hibernate and Bean Validation works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts, updates and deletes done by Hibernate.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in a set of `ConstraintViolation`s. Among other information, the `ConstraintViolation` contains an error description message that can embed the parameter values bundle with the annotation (eg. size limit), and message strings that may be externalized to a `ResourceBundle`.

23.1.1. Adding Bean Validation

To enable Hibernate's Bean Validation integration, simply add a Bean Validation provider (preferably Hibernate Validation 4) on your classpath.

23.1.2. Configuration

By default, no configuration is necessary.

The `Default` group is validated on entity insert and update and the database model is updated accordingly based on the `Default` group as well.

You can customize the Bean Validation integration by setting the validation mode. Use the `javax.persistence.validation.mode` property and set it up for example in your `persistence.xml` file or your `hibernate.cfg.xml` file. Several options are possible:

- `auto` (default): enable integration between Bean Validation and Hibernate (callback and ddl generation) only if Bean Validation is present in the classpath.
- `none`: disable all integration between Bean Validation and Hibernate
- `callback`: only validate entities when they are either inserted, updated or deleted. An exception is raised if no Bean Validation provider is present in the classpath.
- `ddl`: only apply constraints to the database schema when generated by Hibernate. An exception is raised if no Bean Validation provider is present in the classpath. This value is not defined by the Java Persistence spec and is specific to Hibernate.

##

You can use both `callback` and `ddl` together by setting the property to `callback, dll`

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.mode"
                value="callback, ddl"/>
    </properties>
  </persistence-unit>
</persistence>
```

This is equivalent to `auto` except that if no Bean Validation provider is present, an exception is raised.

If you want to validate different groups during insertion, update and deletion, use:

- `javax.persistence.validation.group.pre-persist`: groups validated when an entity is about to be persisted (default to `Default`)
- `javax.persistence.validation.group.pre-update`: groups validated when an entity is about to be updated (default to `Default`)
- `javax.persistence.validation.group.pre-remove`: groups validated when an entity is about to be deleted (default to no group)
- `org.hibernate.validator.group.ddl`: groups considered when applying constraints on the database schema (default to `Default`)

Each property accepts the fully qualified class names of the groups validated separated by a comma (,)

#23.1 Using custom groups for validation

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.group.Default, com.acme.group.Strict"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value="com.acme.group.OnDelete"/>
      <property name="org.hibernate.validator.group.ddl"
                value="com.acme.group.DDL"/>
    </properties>
  </persistence-unit>
</persistence>
```

##

You can set these properties in `hibernate.cfg.xml`, `hibernate.properties` or programmatically.

23.1.3. Catching violations

If an entity is found to be invalid, the list of constraint violations is propagated by the `ConstraintViolationException` which exposes the set of `ConstraintViolations`.

This exception is wrapped in a `RollbackException` when the violation happens at commit time. Otherwise the `ConstraintViolationException` is returned (for example when calling `flush()`. Note that generally, catchable violations are validated at a higher level (for example in Seam / JSF 2 via the JSF - Bean Validation integration or in your business layer by explicitly calling Bean Validation).

An application code will rarely be looking for a `ConstraintViolationException` raised by Hibernate. This exception should be treated as fatal and the persistence context should be discarded (`EntityManager` or `Session`).

23.1.4. Database schema

Hibernate uses Bean Validation constraints to generate an accurate database schema:

- `@NotNull` leads to a not null column (unless it conflicts with components or table inheritance)
- `@Size.max` leads to a `varchar(max)` definition for Strings
- `@Min`, `@Max` lead to column checks (like `value <= max`)
- `@Digits` leads to the definition of precision and scale (ever wondered which is which? It's easy now with `@Digits` :))

These constraints can be declared directly on the entity properties or indirectly by using constraint composition.

For more information check the Hibernate Validator *reference documentation* [http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/].

23.2. Hibernate Search

23.2.1. Description

Full text search engines like Apache Lucene™ are a very powerful technology to bring free text/efficient queries to applications. If suffers several mismatches when dealing with a object domain model (keeping the index up to date, mismatch between the index structure and the domain model, querying mismatch...) Hibernate Search indexes your domain model thanks to a few annotations, takes care of the database / index synchronization and brings you back regular managed objects from free text queries. Hibernate Search is using *Apache Lucene* [http://lucene.apache.org] under the cover.

23.2.2. Integration with Hibernate Annotations

Hibernate Search integrates with Hibernate Core transparently provided that the Hibernate Search jar is present on the classpath. If you do not wish to automatically register Hibernate Search event listeners, you can set `hibernate.search.autoregister_listeners` to false. Such a need is very uncommon and not recommended.

Check the Hibernate Search *reference documentation* [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html/] for more information.

#/##

####### Hibernate ## ##################################### # ## ## ## <one-to-many> ##### # # ## # #################################### ## # <composite-element> ################ # Hibernate ## ################################# ################### ####################### ########

24.1.

Hibernate ## ###############

- ###
- ##) ########################### ##
- ##############################) ### ## ###############################

#######################################/####################################

24.2.

Parent ## Child ##### <one-to-many> ############

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

#############

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate #### SQL #######:

- c ############## INSERT
- p ## c ########## UPDATE

################ parent_id ######## NOT NULL ######################## not-null="true" ######### null #################:

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set
>
```

##################

############## p ## c ########## parent_id) # Child ########################## INSERT ################################## Child #################

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(## Child #### parent ###################)

Child ## inverse

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

############### Child ##############

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

SQL # INSERT

############ Parent # addChild() ############

```
public void addChild(Child c) {
```

```
    c.setParent(this);
    children.add(c);
}
```

Child ####################

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

24.3.

save()

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

#########################

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

Parent ## p
##########

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

##########

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

######## c ######## p ###################### NOT NULL ################ Child # delete() ##############

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

############ Child ###################################### Child ################## ##### cascade="all-delete-orphan" ##############

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

################ inverse="true" ## ## setParent() ### ##########

24.4. ###### unsaved-value

Suppose we loaded up a `Parent` in one `Session`, made some changes in a UI action and wanted to persist these changes in a new session by calling `update()`. The `Parent` will contain a collection of children and, since the cascading update is enabled, Hibernate needs to know which children are newly instantiated and which represent existing rows in the database. We will also assume that both `Parent` and `Child` have generated identifier properties of type `Long`. Hibernate will use the identifier and version/timestamp property value to determine which of the children are new. (See ##########.) *In Hibernate3, it is no longer necessary to specify an `unsaved-value` explicitly.*

####### parent # child ##### newChild #######

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Hibernate ########## ## Session ############################## ##### Hibernate ##

24.5.

##
Hibernate ################################

############################## <composite-element> ##############################
1##########################
##

#: Weblog

25.1.

set ########## bag

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
```

```
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

25.2. Hibernate

XML ######################

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>
```

```
        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping
>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>
```

```
</hibernate-mapping
>
```

25.3. Hibernate

######## Hibernate ################################

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
```

```
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
```

```
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
```

```
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime
> :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);
```

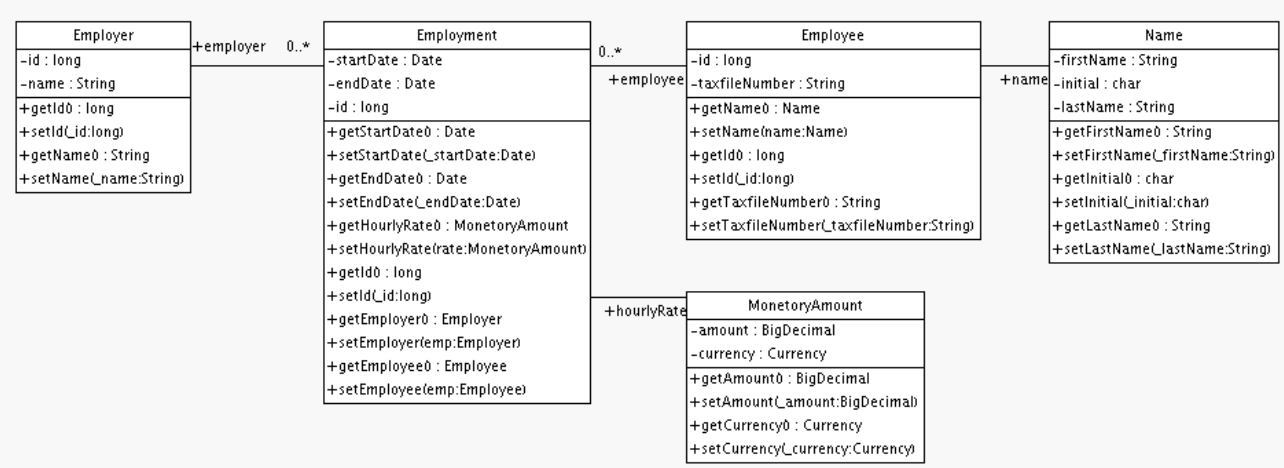
```
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

##########

##############################

26.1. ###/###

Employer # Employee ################################## # Employment # ############## 2###



##################

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence"
>employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence"
>employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
```

```
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence"
>employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping
>
```

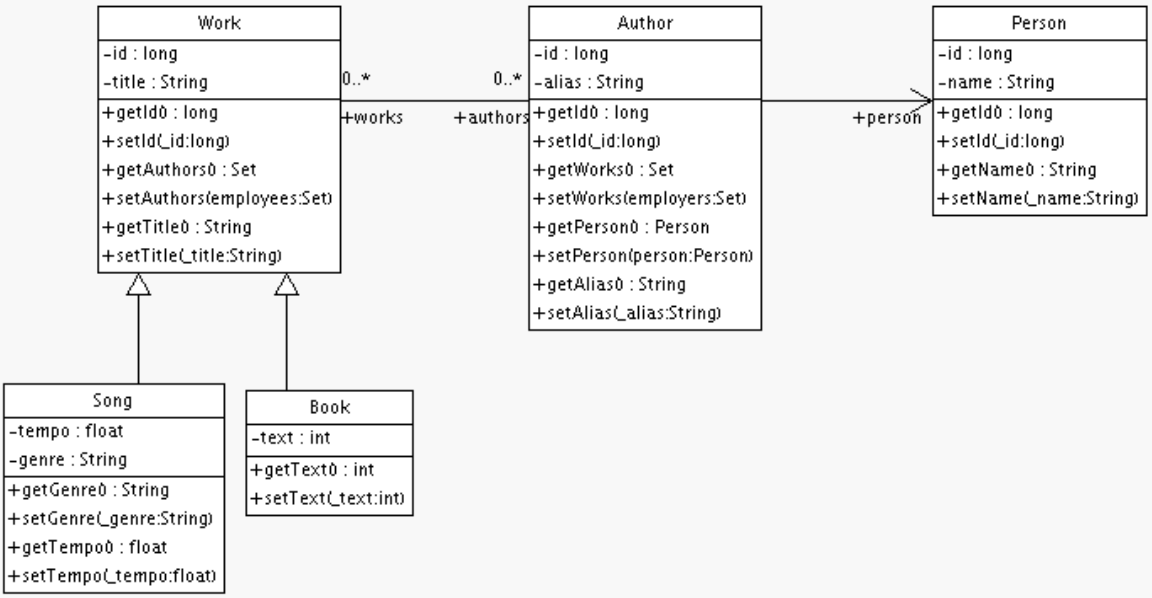
SchemaExport ##################

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
```

```
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

26.2. ##/##

Work # Author ### Person ######################## Work # Author ################## Author # Person ######################## Author # Person #################



##################################:

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>
```

```
        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping
>
```

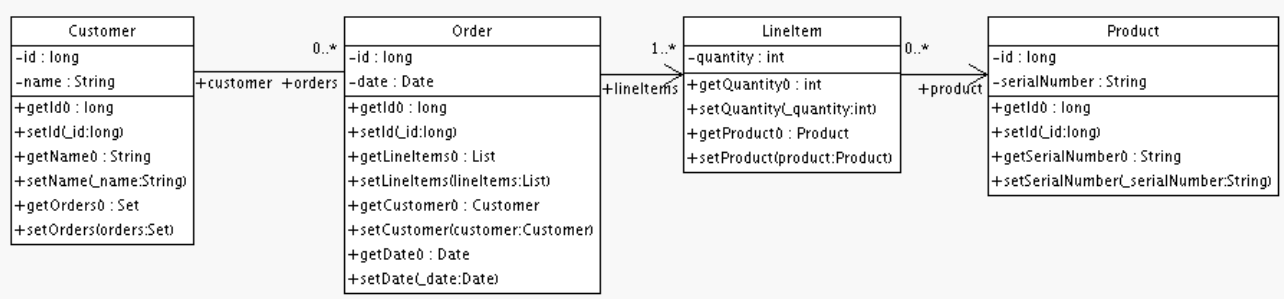
#########4############# works # authors , persons ########################## author_work ########################## SchemaExport #################

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
```

```
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

26.3. ##/##/##

Customer # Order # LineItem # Product ##################### Customer # Order ##### ###### Order / LineItem / Product ################## LineItem ## Order # Product ### ############################ Hibernate



############

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
```

```
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping
>
```

customers # orders # line_items # products ##############################
line_items ############################

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

26.4.

########## Hibernate ## Hibernate ##### ####### `test` ##########

26.4.1.

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula
>name</formula>
        <formula
>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula
>name</formula>
        <formula
>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class
>
```

26.4.2.

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
```

```
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>
```

```
    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class
>
```

26.4.3.

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
```

```
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula
>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula
>org</formula>
        </many-to-many>
    </set>
</class>
```

26.4.4. discrimination

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
```

```
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class
>
```

26.4.5.

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>
```

```
    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class
>
```

##########

############ <component> ############

    `street` ##### `suburb` ##### `state` #### `postcode` ############## `Address` #########
##

########################

    Hibernate ##
#################

#############

    ######################## `<natural-id>` ###################################
`equals()` # `hashCode()` #########

#################################

    ############################# `com.eg.Foo` ##### `com/eg/Foo.hbm.xml` ##########
##############################

######################

    #################################

#####################

    #### ANSI ##### SQL ###
#############

##############

    JDBC ################### "?" ###
#############################

JDBC ##################

    Hibernate ########### JDBC ##
####################### `org.hibernate.connection.ConnectionProvider` #######
#########

################

    ############## Java ##
################# `org.hibernate.UserType` ################# Hibernate ########
#######################

############# JDBC #############

    In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. Do not assume, however, that JDBC is necessarily faster. Please wait until you *know* something is a bottleneck. If you need to use direct JDBC, you can open a Hibernate `Session`, wrap your JDBC operation as a `org.hibernate.jdbc.Work` object and using that JDBC connection. This way you can still use the same transaction strategy and underlying connection provider.

`Session` ###############

Session ##
##

3##############################

/ ##### Bean ################### / JSP ######## Bean
################################### Session ########### `Session.merge()` #
`Session.saveOrUpdate()` ############################

2##############################

############
##############################1#######unit of work#####################
##################/###
###########2###
JDBC
####################################1## Session ##########################
############

########################

`Transaction` ########## `Session`
################ Hibernate ##
`Session.load()`
`Session.get()` #############

########################

##
`lazy="false"`
############################## `left join fetch` #########

############################ ################## *(open session in view)* ###########
######## *(assembly phase)* ########

Hibernate # *Data Transfer Objects* (DTO) ######################### EJB #########
DTO #2######### 1########## Bean ####################2####################
############################ DTO ############################ Hibernate ##1##
##
##
Hibernate

Hibernate ############################

##########Hibernate ##################### *DAO # Thread Local Session* ##########
`UserType` # Hibernate ################ JDBC
################################5############################

########################

##2#
##
#####

Database Portability Considerations

28.1. Portability Basics

One of the selling points of Hibernate (and really Object/Relational Mapping as a whole) is the notion of database portability. This could mean an internal IT user migrating from one database vendor to another, or it could mean a framework or deployable application consuming Hibernate to simultaneously target multiple database products by their users. Regardless of the exact scenario, the basic idea is that you want Hibernate to help you run against any number of databases without changes to your code, and ideally without any changes to the mapping metadata.

28.2. Dialect

The first line of portability for Hibernate is the dialect, which is a specialization of the `org.hibernate.dialect.Dialect` contract. A dialect encapsulates all the differences in how Hibernate must communicate with a particular database to accomplish some task like getting a sequence value or structuring a SELECT query. Hibernate bundles a wide range of dialects for many of the most popular databases. If you find that your particular database is not among them, it is not terribly difficult to write your own.

28.3. Dialect resolution

Originally, Hibernate would always require that users specify which dialect to use. In the case of users looking to simultaneously target multiple databases with their build that was problematic. Generally this required their users to configure the Hibernate dialect or defining their own method of setting that value.

Starting with version 3.2, Hibernate introduced the notion of automatically detecting the dialect to use based on the `java.sql.DatabaseMetaData` obtained from a `java.sql.Connection` to that database. This was much better, expect that this resolution was limited to databases Hibernate know about ahead of time and was in no way configurable or overrideable.

Starting with version 3.3, Hibernate has a fare more powerful way to automatically determine which dialect to should be used by relying on a series of delegates which implement the `org.hibernate.dialect.resolver.DialectResolver` which defines only a single method:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```

The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies `org.hibernate.exception.JDBCConnectionException` as possibly being thrown. A JDBCConnectionException here is interpreted to imply a "non transient" (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver.

The cool part about these resolvers is that users can also register their own custom resolvers which will be processed ahead of the built-in Hibernate ones. This might be useful in a number of different situations: it allows easy integration for auto-detection of dialects beyond those shipped with HIbernate itself; it allows you to specify to use a custom dialect when a particular database is recognized; etc. To register one or more resolvers, simply specify them (seperated by commas, tabs or spaces) using the 'hibernate.dialect_resolvers' configuration setting (see the `DIALECT_RESOLVERS` constant on `org.hibernate.cfg.Environment`).

28.4. Identifier generation

When considering portability between databases, another important decision is selecting the identifier generation stratagy you want to use. Originally Hibernate provided the *native* generator for this purpose, which was intended to select between a *sequence*, *identity*, or *table* strategy depending on the capability of the underlying database. However, an insidious implication of this approach comes about when targtetting some databases which support *identity* generation and some which do not. *identity* generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value; it is what is known as a post-insert generation strategy becauase the insert must actually happen before we can know the identifier value. Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context it must then issue the insert immediately when the users requests the entitiy be associated with the session (like via save() e.g.) regardless of current transactional semantics.

##

Hibernate was changed slightly once the implication of this was better understood so that the insert is delayed in cases where that is feasible.

The underlying issue is that the actual semanctics of the application itself changes in these cases.

Starting with version 3.2.3, Hibernate comes with a set of *enhanced* [http://in.relation.to/2082.lace] identifier generators targetting portability in a much different way.

##

There are specifically 2 bundled *enhanced*generators:

- `org.hibernate.id.enhanced.SequenceStyleGenerator`
- `org.hibernate.id.enhanced.TableGenerator`

The idea behind these generators is to port the actual semantics of the identifer value generation to the different databases. For example, the `org.hibernate.id.enhanced.SequenceStyleGenerator` mimics the behavior of a sequence on databases which do not support sequences by using a table.

28.5. Database functions

##

This is an area in Hibernate in need of improvement. In terms of portability concerns, this function handling currently works pretty well from HQL; however, it is quite lacking in all other aspects.

SQL functions can be referenced in many ways by users. However, not all databases support the same set of functions. Hibernate, provides a means of mapping a *logical* function name to a delegate which knows how to render that particular function, perhaps even using a totally different physical function call.

####

Technically this function registration is handled through the `org.hibernate.dialect.function.SQLFunctionRegistry` class which is intended to allow users to provide custom function definitions without having to provide a custom dialect. This specific behavior is not fully completed as of yet.

It is sort of implemented such that users can programatically register functions with the `org.hibernate.cfg.Configuration` and those functions will be recognized for HQL.

28.6. Type mappings

This section scheduled for completion at a later date...

References

[PoEAA] *Patterns of Enterprise Application Architecture*. 0-321-12742-0. # Fowler Martin [FAMILY Given]. #### © 2003 Pearson Education, Inc.. Addison-Wesley Publishing Company.

[JPwH] *Java Persistence with Hibernate*. Second Edition of Hibernate in Action. 1-932394-88-5. *http://www.manning.com/bauer2* . # Bauer Christian [FAMILY Given] # King Gavin [FAMILY Given]. #### © 2007 Manning Publications Co.. Manning Publications Co..