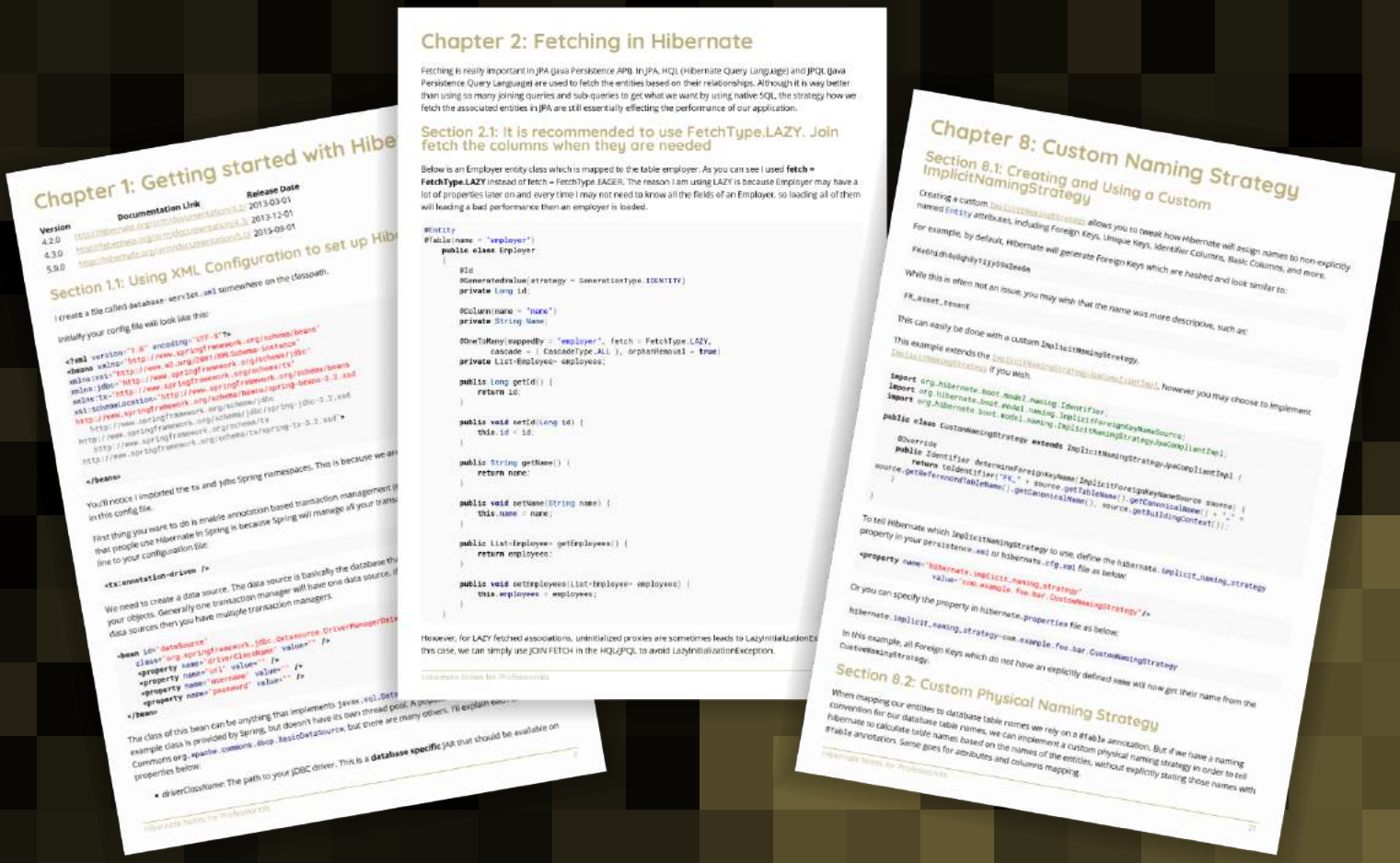
Hibernate

Notes for Professionals



30+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/HibernateBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Hibernate

Version	Documentation Link	Release Date
4.2.0	http://hibernate.org/orm/documentation/4.2/	2013-03-01
4.3.0	http://hibernate.org/orm/documentation/4.3/	2013-12-01
5.0.0	http://hibernate.org/orm/documentation/5.0/	2015-09-01

Section 1.1: Using XML Configuration to set up Hibernate

I create a file called `database-servlet.xml` somewhere on the classpath.

Initially your config file will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
   http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd
   http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">

</beans>
```

You'll notice I imported the `tx` and `jdbc` Spring namespaces. This is because we are going to use them quite heavily in this config file.

First thing you want to do is enable annotation based transaction management (`@Transactional`). The main reason that people use Hibernate in Spring is because Spring will manage all your transactions for you. Add the following line to your configuration file:

```
<tx:annotation-driven />
```

We need to create a data source. The data source is basically the database that Hibernate is going to use to persist your objects. Generally one transaction manager will have one data source. If you want Hibernate to talk to multiple data sources then you have multiple transaction managers.

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="" />
    <property name="url" value="" />
    <property name="username" value="" />
    <property name="password" value="" />
</bean>
```

The class of this bean can be anything that implements `javax.sql.DataSource` so you could write your own. This example class is provided by Spring, but doesn't have its own thread pool. A popular alternative is the Apache Commons `org.apache.commons.dbcp.BasicDataSource`, but there are many others. I'll explain each of the properties below:

- *driverClassName*: The path to your JDBC driver. This is a **database specific** JAR that should be available on

your classpath. Ensure that you have the most up to date version. If you are using an Oracle database, you'll need a OracleDriver. If you have a MySQL database, you'll need a MySQLDriver. See if you can find the driver you need here but a quick google should give you the correct driver.

- *url*: The URL to your database. Usually this will be something like `jdbc\:oracle\:thin\:\path\to\your\database` or `jdbc:mysql://path/to/your/database`. If you google around for the default location of the database you are using, you should be able to find out what this should be. If you are getting a `HibernateException` with the message `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not set` and you are following this guide, there is a 90% chance that your URL is wrong, a 5% chance that your database isn't started and a 5% chance that your username/password is wrong.
- *username*: The username to use when authenticating with the database.
- *password*: The password to use when authenticating with the database.

The next thing, is to set up the `SessionFactory`. This is the thing that Hibernate uses to create and manage your transactions, and actually talks to the database. It has quite a few configuration options that I will try to explain below.

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="au.com.project />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.use_sql_comments">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
        </props>
    </property>
</bean>
```

- *dataSource*: Your data source bean. If you changed the Id of the dataSource, set it here.
- *packagesToScan*: The packages to scan to find your JPA annotated objects. These are the objects that the session factory needs to manage, will generally be POJO's and annotated with `@Entity`. For more information on how to set up object relationships in Hibernate see here.
- *annotatedClasses* (not shown): You can also provide a list of classes for Hibernate to scan if they are not all in the same package. You should use either `packagesToScan` or `annotatedClasses` but not both. The declaration looks like this:

```
<property name="annotatedClasses">
    <list>
        <value>foo.bar.package.model.Person</value>
        <value>foo.bar.package.model.Thing</value>
    </list>
</property>
```

- *hibernateProperties*: There are a myriad of these all lovingly documented here. The main ones you will be using are as follows:
- *hibernate.hbm2ddl.auto*: One of the hottest Hibernate questions details this property. See it for more info. I generally use validate, and set up my database using either SQL scripts (for an in-memory), or create the database beforehand (existing database).
- *hibernate.show_sql*: Boolean flag, if true Hibernate will print all the SQL it generates to `stdout`. You can also configure your logger to show you the values that are being bound to the queries by setting

`log4j.logger.org.hibernate.type=TRACE log4j.logger.org.hibernate.SQL=DEBUG` in your log manager (I use log4j).
- *hibernate.format_sql*: Boolean flag, will cause Hibernate to pretty print your SQL to stdout.
- *hibernate.dialect* (Not shown, for good reason): A lot of old tutorials out there show you how to set the Hibernate dialect that it will use to communicate to your database. Hibernate **can** auto-detect which dialect to use based on the JDBC driver that you are using. Since there are about 3 different Oracle dialects and 5 different MySQL dialects, I'd leave this decision up to Hibernate. For a full list of dialects Hibernate supports see here.

The last 2 beans you need to declare are:

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"
   id="PersistenceExceptionTranslator" />

<bean id="transactionManager"
   class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

The `PersistenceExceptionTranslator` translates database specific `HibernateException` or `SQLExceptions` into Spring exceptions that can be understood by the application context.

The `TransactionManager` bean is what controls the transactions as well as roll-backs.

Note: You should be autowiring your `SessionFactory` bean into your DAO's.

Section 1.2: Simple Hibernate example using XML

To set up a simple hibernate project using XML for the configurations you need 3 files, hibernate.cfg.xml, a POJO for each entity, and a EntityName.hbm.xml for each entity. Here is an example of each using MySQL:

hibernate.cfg.xml

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

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/DBSchemaName
   </property>
   <property name="hibernate.connection.username">
      testUserName
   </property>
   <property name="hibernate.connection.password">
      testPassword
   </property>

   <!-- List of XML mapping files -->
```

```
   <mapping resource="HibernatePractice/Employee.hbm.xml"/>

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

DBSchemaName, testUserName, and testPassword would all be replaced. Make sure to use the full resource name if it is in a package.

Employee.java

```
package HibernatePractice;

public class Employee {
    private int id;
    private String firstName;
    private String middleName;
    private String lastName;

    public Employee(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getMiddleName(){
        return middleName;
    }
    public void setMiddleName(String middleName){
        this.middleName = middleName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Employee.hbm.xml

```
<hibernate-mapping>
   <class name="HibernatePractice.Employee" table="employee">
      <meta attribute="class-description">
         This class contains employee information.
      </meta>
      <id name="id" type="int" column="empolyee_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="middleName" column="middle_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
```

```
   </class>
</hibernate-mapping>
```

Again, if the class is in a package use the full class name packageName.className.

After you have these three files you are ready to use hibernate in your project.

Section 1.3: XML-less Hibernate configuration

This example has been taken from here

```
package com.reborne.SmartHibernateConnector.utils;

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

public class LiveHibernateConnector implements IHibernateConnector {

    private String DB_DRIVER_NAME = "";
    private String DB_URL = "jdbc:h2:~/liveDB;MV_STORE=FALSE;MVCC=FALSE";
    private String DB_USERNAME = "sa";
    private String DB_PASSWORD = "";
    private String DIALECT = "org.hibernate.dialect.H2Dialect";
    private String HBM2DLL = "create";
    private String SHOW_SQL = "true";

    private static Configuration config;
    private static SessionFactory sessionFactory;
    private Session session;

    private boolean CLOSE_AFTER_TRANSACTION = false;

    public LiveHibernateConnector() {

        config = new Configuration();

        config.setProperty("hibernate.connector.driver_class",          DB_DRIVER_NAME);
        config.setProperty("hibernate.connection.url",                  DB_URL);
        config.setProperty("hibernate.connection.username",          DB_USERNAME);
        config.setProperty("hibernate.connection.password",          DB_PASSWORD);
        config.setProperty("hibernate.dialect",                      DIALECT);
        config.setProperty("hibernate.hbm2dll.auto",                  HBM2DLL);
        config.setProperty("hibernate.show_sql",                     SHOW_SQL);

        /*
         * Config connection pools
         */

        config.setProperty("connection.provider_class",
"org.hibernate.connection.C3P0ConnectionProvider");
        config.setProperty("hibernate.c3p0.min_size", "5");
        config.setProperty("hibernate.c3p0.max_size", "20");
        config.setProperty("hibernate.c3p0.timeout", "300");
        config.setProperty("hibernate.c3p0.max_statements", "50");
        config.setProperty("hibernate.c3p0.idle_test_period", "3000");


        /**
         * Resource mapping
```

```
         */

//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);

        sessionFactory = config.buildSessionFactory();
    }


    public HibWrapper openSession() throws HibernateException {
        return new HibWrapper(getOrCreateSession(), CLOSE_AFTER_TRANSACTION);
    }


    public Session getOrCreateSession() throws HibernateException {
        if (session == null) {
            session = sessionFactory.openSession();
        }
        return session;
    }

    public void reconnect() throws HibernateException {
        this.sessionFactory = config.buildSessionFactory();
    }


}
```

Please note, that with latest Hibernate this approach doesn't work well (Hibernate 5.2 release still allow this configuration)

Chapter 2: Fetching in Hibernate

Fetching is really important in JPA (Java Persistence API). In JPA, HQL (Hibernate Query Language) and JPQL (Java Persistence Query Language) are used to fetch the entities based on their relationships. Although it is way better than using so many joining queries and sub-queries to get what we want by using native SQL, the strategy how we fetch the associated entities in JPA are still essentially effecting the performance of our application.

Section 2.1: It is recommended to use FetchType.LAZY. Join fetch the columns when they are needed

Below is an Employer entity class which is mapped to the table employer. As you can see I used **fetch = FetchType.LAZY** instead of fetch = FetchType.EAGER. The reason I am using LAZY is because Employer may have a lot of properties later on and every time I may not need to know all the fields of an Employer, so loading all of them will leading a bad performance then an employer is loaded.

```
@Entity
@Table(name = "employer")
    public class Employer
    {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

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

        @OneToMany(mappedBy = "employer", fetch = FetchType.LAZY,
               cascade = { CascadeType.ALL }, orphanRemoval = true)
        private List<Employee> employees;

        public Long getId() {
            return id;
        }

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

        public String getName() {
            return name;
        }

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

        public List<Employee> getEmployees() {
            return employees;
        }

        public void setEmployees(List<Employee> employees) {
            this.employees = employees;
        }
    }
```

However, for LAZY fetched associations, uninitialized proxies are sometimes leads to LazyInitializationException. In this case, we can simply use JOIN FETCH in the HQL/JPQL to avoid LazyInitializationException.

```
SELECT Employer employer FROM Employer
       LEFT JOIN FETCH employer.name
       LEFT JOIN FETCH employer.employee employee
       LEFT JOIN FETCH employee.name
       LEFT JOIN FETCH employer.address
```

Chapter 3: Hibernate Entity Relationships using Annotations

Annotation	Details
`@OneToOne`	Specifies a one to one relationship with a corresponding object.
`@OneToMany`	Specifies a single object that maps to many objects.
`@ManyToOne`	Specifies a collection of objects that map to a single object.
`@Entity`	Specifies an object that maps to a database table.
`@Table`	Specifies which database table this object maps too.
`@JoinColumn`	Specifies which column a foregin key is stored in.
`@JoinTable`	Specifies an intermediate table that stores foreign keys.

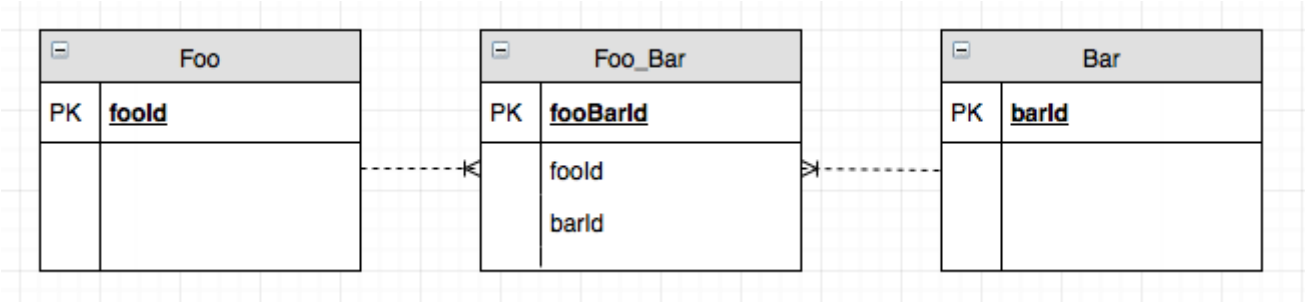
Section 3.1: Bi-Directional Many to Many using user managed join table object

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<FooBar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany(mappedBy = "foo")
    private List<FooBar> foos;
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId")
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

Specifies a two-way relationship between many `Foo` objects to many `Bar` objects using an intermediate join table that the user manages.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The relationships between `Foo` and `Bar` objects are stored in a table called `FOO_BAR`. There is a `FooBar` object as part of the application.

Commonly used when you want to store extra information on the join object such as the date the relationship was created.

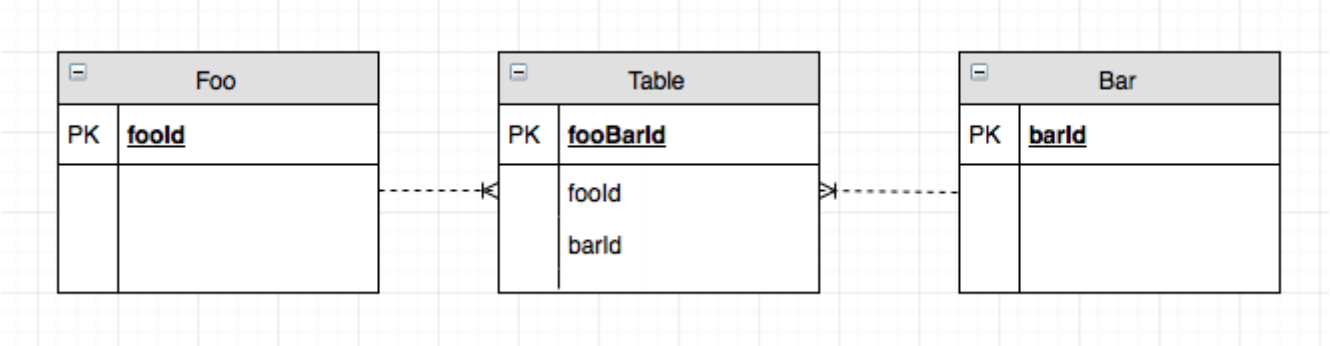


Section 3.2: Bi-Directional Many to Many using Hibernate managed join table

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId"))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="barId"),
        inverseJoinColumns = @JoinColumn(name="fooId"))
    private List<Foo> foos;
}
```

Specifies a relationship between many `Foo` objects to many `Bar` objects using an intermediate join table that Hibernate manages.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The relationships between `Foo` and `Bar` objects are stored in a table called `FOO_BAR`. However this implies that there is no `FooBar` object as part of the application.

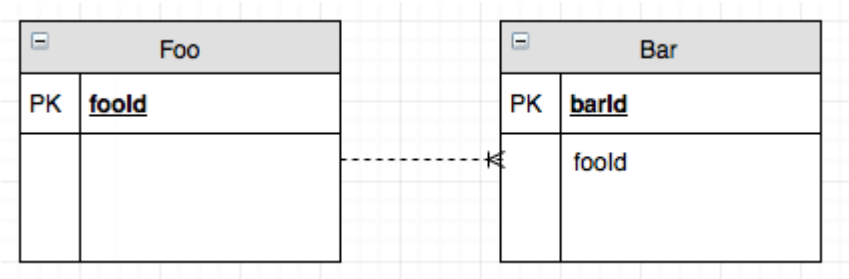


Section 3.3: Bi-directional One to Many Relationship using foreign key mapping

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;
}
```

Specifies a two-way relationship between one `Foo` object to many `Bar` objects using a foreign key.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The foreign key is stored on the `BAR` table in a column called `fooId`.



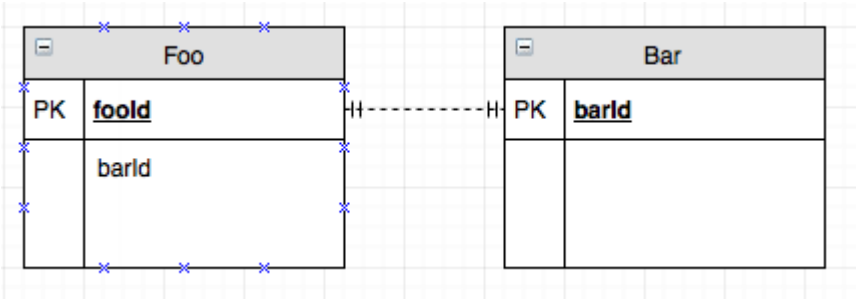
Section 3.4: Bi-Directional One to One Relationship managed by Foo.class

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "barId")
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToOne(mappedBy = "bar")
    private Foo foo;
}
```

Specifies a two-way relationship between one `Foo` object to one `Bar` object using a foreign key.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The foreign key is stored on the `FOO` table in a column called `barId`.

Note that the `mappedBy` value is the field name on the object, not the column name.



Section 3.5: Uni-Directional One to Many Relationship using user managed join table

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId", unique=true))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    //No Mapping specified here.
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId", unique = true)
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

Specifies a one-way relationship between one `Foo` object to many `Bar` objects using an intermediate join table that the user manages.

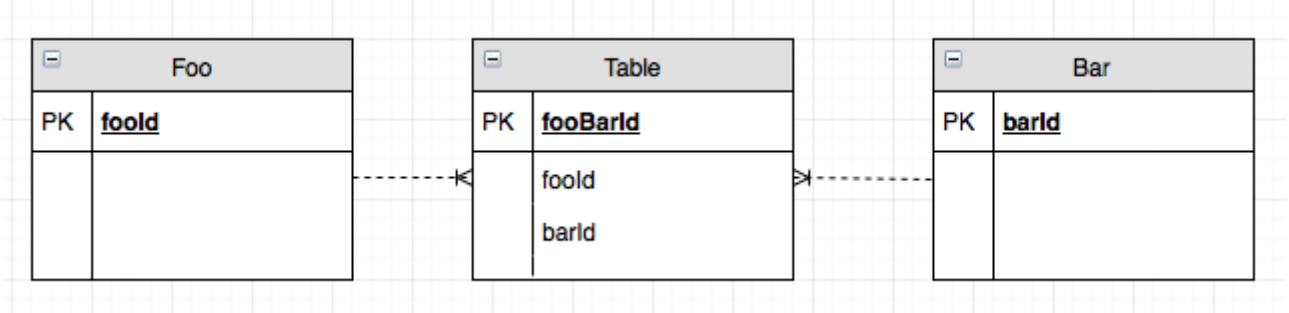
This is similar to a `ManyToMany` relationship, but if you add a `unique` constraint to the target foreign key you can

enforce that it is `OneToMany`.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The relationships between `Foo` and `Bar` objects are stored in a table called `FOO_BAR`. There is a `FooBar` object as part of the application.

Notice that there is no mapping of `Bar` objects back to `Foo` objects. `Bar` objects can be manipulated freely without affecting `Foo` objects.

Very commonly used with Spring Security when setting up a `User` object who has a list of `Role`'s that they can perform. You can add and remove roles to a user without having to worry about cascades deleting `Role`'s.



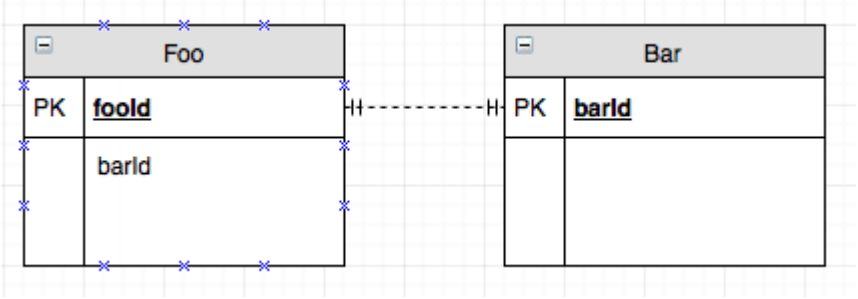
Section 3.6: Uni-directional One to One Relationship

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;
    //No corresponding mapping to Foo.class
}
```

Specifies a one-way relationship between one `Foo` object to one `Bar` object.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`.

Notice that there is no mapping of `Bar` objects back to `Foo` objects. `Bar` objects can be manipulated freely without affecting `Foo` objects.



Chapter 4: HQL

HQL is Hibernate Query Language, it based on SQL and behind the scenes it is changed into SQL but the syntax is different. You use entity/class names not table names and field names not column names. It also allows many shorthands.

Section 4.1: Selecting a whole table

```
hql = "From EntityName";
```

Section 4.2: Select specific columns

```
hql = "Select id, name From Employee";
```

Section 4.3: Include a Where clause

```
hql = "From Employee where id = 22";
```

Section 4.4: Join

```
hql = "From Author a, Book b Where a.id = book.author";
```

Chapter 5: Native SQL Queries

Section 5.1: Simple Query

Assuming you have a handle on the Hibernate `Session` object, in this case named `session`:

```
List<Object[]> result = session.createNativeQuery("SELECT * FROM some_table").list();
for (Object[] row : result) {
    for (Object col : row) {
        System.out.print(col);
    }
}
```

This will retrieve all rows in `some_table` and place them into the `result` variable and print every value.

Section 5.2: Example to get a unique result

```
Object pollAnswered = getCurrentSession().createSQLQuery(
        "select * from TJ_ANSWERED_ASW where pol_id = "+pollId+" and prf_log =
'"+logid+"'").uniqueResult();
```

with this query, you get a unique result when you know the result of the query is always going to be unique.

And if the query returns more than one value, you will get an exception

> org.hibernate.NonUniqueResultException

You also check the details in this link here with more discription

So, please be sure that you know the query will return unique result

Chapter 6: Mapping associations

Section 6.1: One to One Hibernate Mapping

Every Country has one Capital. Every Capital has one Country.

Country.java

```
package com.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "countries")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "national_language")
    private String nationalLanguage;

    @OneToOne(mappedBy = "country")
    private Capital capital;

    //Constructor

    //getters and setters

   }
```

Capital.java

```
package com.entity;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "capitals")
public class Capital {

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

```
    private String name;

    private long population;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "country_id")
    private Country country;

    //Constructor

    //getters and setters

}
```

HibernateDemo.java

```
package com.entity;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateDemo {

public static void main(String ar[]) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Country india = new Country();
    Capital delhi = new Capital();
    delhi.setName("Delhi");
    delhi.setPopulation(357828394);
    india.setName("India");
    india.setNationalLanguage("Hindi");
    delhi.setCountry(india);
    session.save(delhi);
    session.close();
  }

}
```

Chapter 7: Criterias and Projections

Section 7.1: Use Filters

`@Filter` is used as a `WHERE` camp, here some examples

Student Entity

```
@Entity
@Table(name = "Student")
public class Student
{
    /*...*/

    @OneToMany
    @Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and s.id
= study_id)")
    Set<StudentStudy> studies;

    /* getters and setters methods */
}
```

Study Entity

```
@Entity
@Table(name = "Study")
@FilterDef(name = "active")
@Filter(name = "active", condition="state = true")
public class Study
{
    /*...*/

    @OneToMany
    Set<StudentStudy> students;

    @Field
    boolean state;

    /* getters and setters methods */
}
```

StudentStudy Entity

```
@Entity
@Table(name = "StudentStudy")
@Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and s.id =
study_id)")
public class StudentStudy
{
    /*...*/

    @ManytoOne
    Student student;

    @ManytoOne
    Study study;

    /* getters and setters methods */
```

```
}
```

This way, every time the "active" filter is enabled,

-Every query we do on the student entity will return **ALL** Students with **ONLY** their `state = true` studies

-Every query we do on the Study entity will return **ALL** `state = true` studies

-Every query we do on the StudentStudy entiy will return **ONLY** the ones with a `state = true` Study relationship

Pls note that study_id is the name of the field on the sql StudentStudy table

Section 7.2: List using Restrictions

Assuming we have a TravelReview table with City names as column "title"

```
Criteria criteria =
   session.createCriteria(TravelReview.class);
 List review =
   criteria.add(Restrictions.eq("title", "Mumbai")).list();
 System.out.println("Using equals: " + review);
```

We can add restrictions to the criteria by chaining them as follows:

```
List reviews = session.createCriteria(TravelReview.class)
   .add(Restrictions.eq("author", "John Jones"))
   .add(Restrictions.between("date",fromDate,toDate))
   .add(Restrictions.ne("title","New York")).list();
```

Section 7.3: Using Projections

Should we wish to retrieve only a few columns, we can use the Projections class to do so. For example, the following code retrieves the title column

```
// Selecting all title columns
 List review = session.createCriteria(TravelReview.class)
       .setProjection(Projections.property("title"))
   .list();
 // Getting row count
 review = session.createCriteria(TravelReview.class)
   .setProjection(Projections.rowCount())
   .list();
 // Fetching number of titles
 review = session.createCriteria(TravelReview.class)
   .setProjection(Projections.count("title"))
   .list();
```

Chapter 8: Custom Naming Strategy

Section 8.1: Creating and Using a Custom ImplicitNamingStrategy

Creating a custom `ImplicitNamingStrategy` allows you to tweak how Hibernate will assign names to non-explicitly named `Entity` attributes, including Foreign Keys, Unique Keys, Identifier Columns, Basic Columns, and more.

For example, by default, Hibernate will generate Foreign Keys which are hashed and look similar to:

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

While this is often not an issue, you may wish that the name was more descriptive, such as:

```
FK_asset_tenant
```

This can easily be done with a custom `ImplicitNamingStrategy`.

This example extends the `ImplicitNamingStrategyJpaCompliantImpl`, however you may choose to implement `ImplicitNamingStrategy` if you wish.

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.ImplicitForeignKeyNameSource;
import org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl;

public class CustomNamingStrategy extends ImplicitNamingStrategyJpaCompliantImpl {

    @Override
    public Identifier determineForeignKeyName(ImplicitForeignKeyNameSource source) {
        return toIdentifier("FK_" + source.getTableName().getCanonicalName() + "_" +
source.getReferencedTableName().getCanonicalName(), source.getBuildingContext());
    }

}
```

To tell Hibernate which `ImplicitNamingStrategy` to use, define the `hibernate.implicit_naming_strategy` property in your `persistence.xml` or `hibernate.cfg.xml` file as below:

```
<property name="hibernate.implicit_naming_strategy"
                value="com.example.foo.bar.CustomNamingStrategy"/>
```

Or you can specify the property in `hibernate.properties` file as below:

```
hibernate.implicit_naming_strategy=com.example.foo.bar.CustomNamingStrategy
```

In this example, all Foreign Keys which do not have an explicitly defined `name` will now get their name from the `CustomNamingStrategy`.

Section 8.2: Custom Physical Naming Strategy

When mapping our entities to database table names we rely on a `@Table` annotation. But if we have a naming convention for our database table names, we can implement a custom physical naming strategy in order to tell hibernate to calculate table names based on the names of the entities, without explicitly stating those names with `@Table` annotation. Same goes for attributes and columns mapping.

For example, our entity name is:

```
ApplicationEventLog
```

And our table name is:

```
application_event_log
```

Our Physical naming strategy needs to convert from entity names that are camel case to our db table names which are snake case. We can achieve this by extending hibernate's `PhysicalNamingStrategyStandardImpl`:

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;

public class PhysicalNamingStrategyImpl extends PhysicalNamingStrategyStandardImpl {

    private static final long serialVersionUID = 1L;
    public static final PhysicalNamingStrategyImpl INSTANCE = new PhysicalNamingStrategyImpl();

    @Override
    public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    @Override
    public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    protected static String addUnderscores(String name) {
        final StringBuilder buf = new StringBuilder(name);
        for (int i = 1; i < buf.length() - 1; i++) {
            if (Character.isLowerCase(buf.charAt(i - 1)) &&
                    Character.isUpperCase(buf.charAt(i)) &&
                    Character.isLowerCase(buf.charAt(i + 1))) {
                buf.insert(i++, '_');
            }
        }
        return buf.toString().toLowerCase(Locale.ROOT);
    }
}
```

We are overriding default behavior of methods `toPhysicalTableName` and `toPhysicalColumnName` to apply our db naming convention.

In order to use our custom implementation we need to define `hibernate.physical_naming_strategy` property and give it the name of our `PhysicalNamingStrategyImpl` class.

```
hibernate.physical_naming_strategy=com.example.foo.bar.PhysicalNamingStrategyImpl
```

This way we can alleviate our code from `@Table` and `@Column` annotations, so our entity class:

```
@Entity
public class ApplicationEventLog {
    private Date startTimestamp;
    private String logUser;
    private Integer eventSuccess;
```

```
    @Column(name="finish_dtl")
    private String finishDetails;
}
```

will be correctly be mapped to db table:

```
CREATE TABLE application_event_log (
  ...
  start_timestamp timestamp,
  log_user varchar(255),
  event_success int(11),
  finish_dtl varchar(2000),
  ...
)
```

As seen in the example above, we can still explicitly state the name of the db object if it is not, for some reason, in accordance with our general naming convention: `@Column(name="finish_dtl")`

Chapter 9: Caching

Section 9.1: Enabling Hibernate Caching in WildFly

To enable Second Level Caching for Hibernate in WildFly, add this property to your `persistence.xml` file:

```
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

You may also enable Query Caching with this property:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

WildFly does not require you to define a Cache Provider when enabling Hibernate's Second-Level Cache, as Infinispan is used by default. If you would like to use an alternative Cache Provider, however, you may do so with the `hibernate.cache.provider_class` property.

Chapter 10: Association Mappings between Entities

Section 10.1: One to many association using XML

This is an example of how you would do a one to many mapping using XML. We will use Author and Book as our example and assume an author may have written many books, but each book will only have one author.

Author class:

```
public class Author {
    private int id;
    private String firstName;
    private String lastName;

    public Author(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Book class:

```
public class Book {
    private int id;
    private String isbn;
    private String title;
    private Author author;
    private String publisher;

    public Book() {
        super();
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getIsbn() {
        return isbn;
```

```
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public Author getAuthor() {
        return author;
    }
    public void setAuthor(Author author) {
        this.author = author;
    }
    public String getPublisher() {
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
}
```

Author.hbm.xml:

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

<hibernate-mapping>
   <class name="Author" table="author">
      <meta attribute="class-description">
         This class contains the author's information.
      </meta>
      <id name="id" type="int" column="author_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Book.hbm.xml:

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

<hibernate-mapping>
    <class name="Book" table="book_title">
      <meta attribute="class-description">
         This class contains the book information.
      </meta>
        <id name="id" type="int" column="book_id">
            <generator class="native"/>
        </id>
        <property name="isbn" column="isbn" type="string"/>
        <property name="title" column="title" type="string"/>
```

```
        <many-to-one name="author" class="Author" cascade="all">
            <column name="author"></column>
        </many-to-one>
       <property name="publisher" column="publisher" type="string"/>
    </class>
</hibernate-mapping>
```

What makes the one to many connection is that the Book class contains an Author and the xml has the <many-to-one> tag. The cascade attribute allows you to set how the child entity will be saved/updated.

Section 10.2: OneToMany association

To illustrate relation OneToMany we need 2 Entities e.g. Country and City. One Country has multiple Cities.

In the CountryEntity beloww we define set of cities for Country.

```
@Entity
@Table(name = "Country")
public class CountryEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "COUNTRY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           countryId;

   @Column(name = "COUNTRY_NAME", unique = true, nullable = false, length = 100)
   private String            countryName;

   @OneToMany(mappedBy="country", fetch=FetchType.LAZY)
   private Set<CityEntity> cities = new HashSet<>();

   //Getters and Setters are not shown
}
```

Now the city entity.

```
@Entity
@Table(name = "City")
public class CityEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "CITY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           cityId;

   @Column(name = "CITY_NAME", unique = false, nullable = false, length = 100)
   private String            cityName;

   @ManyToOne(optional=false, fetch=FetchType.EAGER)
   @JoinColumn(name="COUNTRY_ID", nullable=false)
   private CountryEntity country;

   //Getters and Setters are not shown
}
```

Chapter 11: Lazy Loading vs Eager Loading

Section 11.1: Lazy Loading vs Eager Loading

Fetching or loading data can be primarily classified into two types: eager and lazy.

In order to use Hibernate make sure you add the latest version of it to the dependencies section of your pom.xml file:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.1.Final</version>
</dependency>
```

1. Eager Loading And Lazy Loading

The first thing that we should discuss here is what lazy loading and eager loading are:

Eager Loading is a design pattern in which data initialization occurs on the spot. It means that collections are fetched fully at the time their parent is fetched (fetch immediately)

Lazy Loading is a design pattern which is used to defer initialization of an object until the point at which it is needed. This can effectively contribute to application's performance.

2. Using The Different Types Of Loading

Lazy loading can be enabled using the following XML parameter:

```
lazy="true"
```

Let's delve into the example. First we have a User class:

```
public class User implements Serializable {

    private Long userId;
    private String userName;
    private String firstName;
    private String lastName;
    private Set<OrderDetail> orderDetail = new HashSet<>();

    //setters and getters
    //equals and hashcode
    }
```

Look at the Set of orderDetail that we have. Now let's have a look at the **OrderDetail class**:

```
public class OrderDetail implements Serializable {

    private Long orderId;
    private Date orderDate;
    private String orderDesc;
    private User user;

    //setters and getters
    //equals and hashcode
```

```
}
```

The important part that is involved in setting the lazy loading in the `UserLazy.hbm.xml`:

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

This is how the lazy loading is enabled. To disable lazy loading we can simply use: `lazy = "false"` and this in turn will enable eager loading. The following is the example of setting up eager loading in another file User.hbm.xml:

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="false" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

Section 11.2: Scope

For those who haven't played with these two designs, the scope of lazy and eager is within a specific **Session** of *SessionFactory*. *Eager* loads everything instantly, means there is no need to call anything for fetching it. But lazy fetch usually demands some action to retrieve mapped collection/object. This sometimes is problematic getting lazy fetch outside the *session*. For instance, you have a view which shows the detail of the some mapped POJO.

```
@Entity
public class User {
    private int userId;
    private String username;
    @OneToMany
    private Set<Page> likedPage;

    // getters and setters here
}

@Entity
public class Page{
    private int pageId;
    private String pageURL;

    // getters and setters here
}

public class LazzyTest{
    public static void main(String...s){
        SessionFactory sessionFactory = new SessionFactory();
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        User user = session.get(User.class, 1);
        transaction.commit();
        session.close();

        // here comes the lazy fetch issue
        user.getLikedPage();
    }
```

```
}
```

When you will try to get **lazy fetched** outside the *session* you will get the *lazyinitializeException*. This is because by default fetch strategy for all oneToMany or any other relation is *lazy*(call to DB on demand) and when you have closed the session, you have no power to communicate with database. so our code tries to fetch collection of *likedPage* and it throws exception because there is no associated session for rendering DB.

Solution for this is to use:

1. Open Session in View - In which you keep the session open even on the rendered view.
2. `Hibernate.initialize(user.getLikedPage())` before closing session - This tells hibernate to initialize the collection elements

Chapter 12: Enable/Disable SQL log

Section 12.1: Using a logging config file

In the logging configuration file of your choice set the logging of the following packages to the levels shown.:

```
# log the sql statement
org.hibernate.SQL=DEBUG
# log the parameters
org.hibernate.type=TRACE
```

There will probably be some logger specific prefixes that are required.

Log4j config:

```
log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE
```

Spring Boot `application.properties`:

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
```

Logback `logback.xml`:

```
<logger name="org.hibernate.SQL" level="DEBUG"/>
<logger name="org.hibernate.type" level="TRACE"/>
```

Section 12.2: Using Hibernate properties

This will show you the generated SQL, but will not show you the values contained within the queries.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="hibernateProperties">
        <props>
            <!-- show the sql without the parameters -->
            <prop key="hibernate.show_sql">true</prop>
            <!-- format the sql nice -->
            <prop key="hibernate.format_sql">true</prop>
            <!-- show the hql as comment -->
            <prop key="use_sql_comments">true</prop>
        </props>
    </property>
</bean>
```

Section 12.3: Enable/Disable SQL log in debug

Some applications that use Hibernate generate a huge amount of SQL when the application is started. Sometimes it's better to enable/disable the SQL log in specific points when debugging.

To enable, just run this code in your IDE when you are debugging the aplication:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
```

```
    .setLevel(org.apache.log4j.Level.DEBUG)
```

To disable:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.OFF)
```

Chapter 13: Hibernate and JPA

Section 13.1: Relationship between Hibernate and JPA

Hibernate is an implementation of the JPA standard. As such, everything said there is also true for Hibernate.

Hibernate has some extensions to JPA. Also, the way to set up a JPA provider is provider-specific. This documentation section should only contain what is specific to Hibernate.

Chapter 14: Performance tuning

Section 14.1: Use composition instead of inheritance

Hibernate has some strategies of inheritance. The `JOINED` inheritance type do a JOIN between the child entity and parent entity.

The problem with this approach is that Hibernate **always** bring the data of all involved tables in the inheritance.

Per example, if you have the entities `Bicycle` and `MountainBike` using the `JOINED` inheritance type:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Bicycle {

}
```

And:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class MountainBike extends Bicycle {

}
```

Any JPQL query that hit `MountainBike` will brings the `Bicycle` data, creating a SQL query like:

```
SELECT mb.*, b.* FROM MountainBike mb JOIN Bicycle b ON b.id = mb.id WHERE ...
```

If you have another parent for `Bicycle` (like `Transport`, per example), this above query will brings the data from this parent too, doing an extra JOIN.

As you can see, this is a kind of `EAGER` mapping too. You don't have the choice to bring only the data of the `MountainBike` table using this inheritance strategy.

The best for performance is use composition instead of inheritance.

To accomplish this, you can mapping the `MountainBike` entity to have a field `bicycle`:

```
@Entity
public class MountainBike {

    @OneToOne(fetchType = FetchType.LAZY)
    private Bicycle bicycle;

}
```

And `Bicycle`:

```
@Entity
public class Bicycle {

}
```

Every query now will bring only the `MountainBike` data by default.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Aleksei Loginov	Chapter 3
BELLIL	Chapter 11
Daniel Käfer	Chapters 5 and 12
Dherik	Chapters 12 and 14
JamesENL	Chapters 1, 3 and 12
Michael Piefel	Chapters 12 and 13
Mitch Talmadge	Chapters 8 and 9
Naresh Kumar	Chapters 1 and 8
Nathaniel Ford	Chapter 5
omkar sirra	Chapter 6
Pritam Banerjee	Chapter 11
Reborn	Chapter 1
rObOtAndChalie	Chapter 2
Saifer	Chapter 7
Sameer Srivastava	Chapter 7
Sandeep Kamath	Chapter 5
StanislavL	Chapter 10
user7491506	Chapters 1, 4 and 10
veljkost	Chapter 8
vicky	Chapter 11

You may also like