**Fetching strategies**

Hibernate uses a fetching strategy to retrieve associated objects if the application needs to navigate the association. Fetch strategies can be declared in the O/R mapping metadata, or over-ridden by a particular HQL or Criteria query.

Hibernate 3 defines the following fetching strategies:

* **Join fetching**: Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
* **Select fetching**: a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.
* **Subselect fetching**: a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.
* **Batch fetching**: an optimization strategy for select fetching. Hibernate retrieves a batch of entity instances or collections in a single SELECT by specifying a list of primary or foreign keys.

Hibernate also distinguishes between:

* **Immediate fetching**: an association, collection or attribute is fetched immediately when the owner is loaded.
* **Lazy collection fetching**: a collection is fetched when the application invokes an operation upon that collection. This is the default for collections.
* **"Extra-lazy" collection fetching**: individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed. It is suitable for large collections.
* **Proxy fetching**: a single-valued association is fetched when a method other than the identifier getter is invoked upon the associated object.
* "**No-proxy" fetching**: a single-valued association is fetched when the instance variable is accessed. Compared to proxy fetching, this approach is less lazy; the association is fetched even when only the identifier is accessed. It is also more transparent, since no proxy is visible to the application. This approach requires buildtime bytecode instrumentation and is rarely necessary.
* **Lazy attribute fetching**: an attribute or single valued association is fetched when the instance variable is accessed. This approach requires buildtime bytecode instrumentation and is rarely necessary.

We have two orthogonal notions here: when is the association fetched and how is it fetched. It is important that you do not confuse them. We use fetch to tune performance. We can use lazy to define a contract for what data is always available in any detached instance of a particular class.

<http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/performance.html>

# Stored Procedure In Hibernate

# MySQL store procedure

Here’s a MySQL store procedure, which accept a stock code parameter and return the related stock data.

DELIMITER $$

**CREATE** **PROCEDURE** `GetStocks`(int\_stockcode **VARCHAR**(20))

**BEGIN**

**SELECT** \* **FROM** stock **WHERE** stock\_code = int\_stockcode;

**END** $$

DELIMITER ;

In MySQL, you can simple call it with a **call** keyword :

**CALL** GetStocks('7277');

### Hibernate call store procedure

In Hibernate, there are three approaches to call a database store procedure.

#### 1. Native SQL – createSQLQuery

You can use **createSQLQuery()** to call a store procedure directly.

Query query = session.createSQLQuery(

"CALL GetStocks(:stockCode)")

.addEntity(Stock.**class**)

.setParameter("stockCode", "7277");

List result = query.list();

**for**(**int** i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}

#### 2. NamedNativeQuery in annotation

Declare your store procedure inside the **@NamedNativeQueries** annotation.

*//Stock.java*

...

@NamedNativeQueries({

@NamedNativeQuery(

name = "callStockStoreProcedure",

query = "CALL GetStocks(:stockCode)",

resultClass = Stock.**class**

)

})

@Entity

@Table(name = "stock")

**public** **class** Stock **implements** java.io.Serializable {

...

Call it with **getNamedQuery()**.

Query query = session.getNamedQuery("callStockStoreProcedure")

.setParameter("stockCode", "7277");

List result = query.list();

**for**(**int** i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}

#### 3. sql-query in XML mapping file

Declare your store procedure inside the “**sql-query**” tag.

*<!-- Stock.hbm.xml -->*

...

**<hibernate-mapping>**

**<class** name="com.mkyong.common.Stock" table="stock" ...**>**

**<id** name="stockId" type="java.lang.Integer"**>**

**<column** name="STOCK\_ID" **/>**

**<generator** class="identity" **/>**

**</id>**

**<property** name="stockCode" type="string"**>**

**<column** name="STOCK\_CODE" length="10" not-null="true" unique="true" **/>**

**</property>**

...

**</class>**

**<sql-query** name="callStockStoreProcedure"**>**

**<return** alias="stock" class="com.mkyong.common.Stock"**/>**

<![CDATA[CALL GetStocks(:stockCode)]]>

**</sql-query>**

**</hibernate-mapping>**

Call it with **getNamedQuery()**.

Query query = session.getNamedQuery("callStockStoreProcedure")

.setParameter("stockCode", "7277");

List result = query.list();

**for**(**int** i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}

### Conclusion

The above three approaches are doing the same thing, call a store procedure in database. There are not much big different between the three approaches, which method you choose is depend on your personal prefer.