# Versioning & Optimistic Locking in Hibernate

by **Jim White** | Jun 23, 2010

*By Jim White (Directory of Training and Instructor)*

A few weeks ago while teaching Hibernate, a student (thanks Dan) asked me about whether version numbers can be generated by the database. The answer is – Yes, in some cases. I thought the question and an expanded discussion of Hibernate's versioning property would be a good topic for this week's blog entry.

## The <version> property (or @Version annotation)

For those of you that use Hibernate today, you are probably aware that Hibernate can provide optimistic locking through a version property on your persistent objects. Furthermore, the version property is automatically managed by Hibernate.

To specify a version for your persistent object, simply add a version (or timestamp) property in your mapping file and add a version attribute to your persistent class. Here is an example on a Product class.

```
public class Product {
private long id;
private String name;
private String model;
private double cost;
  private long version;
...
```

```
}
<hibernate-mapping package="com.intertech.domain">
<class name="Product">
<id name="id">
<generator class="native" />
</id>
<version name="version" type="long" />
<property name="name" not-null="true" />
<property name="model" not-null="true" />
<property name="cost" type="double"/>
</class>
</hibernate-mapping>
```

For those using annotations, just mark the version property's getter in the class with the @Version annotation.

```
@Version
public long getVersion() {
return version;
}
```

The version property on the class can be numeric (short, int, or long) or timestamp or calendar.  However, another configuration element is provided for timestamps or calendars (covered below).

With automatic versioning (and version set to a numeric), Hibernate will use the version number to check that the row has not been updated since the last time it was retrieved when updating the persistent object (Product in this example).  Notice the extra where "and version = X" clause below, when a product from above is updated.

```
Hibernate:
/* update
com.intertech.domain.Product */ update
Product
```

```
set
version=?,
name=?,
model=?,
cost=?
where
id=?
and version=?
```

If the version has been updated, Hibernate throws a StaleObjectStateException and the transaction (and any persistent object changes) rollback.  If the update commits without issue, Hibernate also increments the version number automatically (note the “set version = ?” as part of the update clause above).

## Hibernate versus the Database Managing the Version

Under this configuration and in this example, Hibernate (from within your Java application) is managing the version number.  However, some DBA’s may prefer to have the version controlled by database rather than Hibernate, especially when the database may be a timestamp and the application is distributed to multiple systems (maybe even timezones).  If the Hibernate application is on different systems that are not time synchronized, the timestamp generated by each system might be different thereby generating improper optimistic lock success or failures.  To inform Hibernate to have the database manage the version, include a generated=”always” attribute in the Hibernate mapping of the persistent object.

```
<hibernate-mapping package=”com.intertech.domain”>
<class name=”Product”>
<id name=”id”>
<generator class=”native” />
</id>
<version name=”version” type=”long” generated=”always”/>
<property name=”name” not-null=”true” />
<property name=”model” not-null=”true” />
<property name=”cost” type=”double”/>
</class>
</hibernate-mapping>
```

Hibernate will now defer to the database to generate the necessary version and update the persistent object's row accordingly. A warning is in order however. This will cause Hibernate to issue an extra SQL to get the version number back from the database after a successful update or insert. Note the extra SQL issued below after a successful update of a product (Hibernate even comments it to indicate why it is needed). You might also note that the version number is not set as above when updating the product. Again, Hibernate is now totally deferring to the database to manage this property.

Of important note, not all dialects (i.e. databases) support the managing of the version number. HSQLDB, for example, does not allow the version number to be controlled from the database.

```
Hibernate:
/* update
com.intertech.domain.Product */ update
Product
set
name=?,
model=?,
cost=?
where
id=?
and version=?
Hibernate:
/* get generated state com.intertech.domain.Product */ select
product_.version as version8_
from
Product product_
where
product_.id=?
```

## Timestamps for Version

In many situations, your DBA or organization may prefer a timestamp to be used as the version indicator. In theory, the timestamp is a little less safe than a version number. This is because a timestamp can, in theory, be updated by two transactions at the exact same time (depends on the precision of the database) and allow both to update causing a last in wins scenario (violating the optimistic lock). It can also be argued that timestamps also take up more

space in the database.  Depending on the dialect, the use of a timestamp may incur an additional hit of the database to get the timestamp.  However, on the plus side, timestamps provide useful information about the “when” a row was last altered.  In cases of auditing or record tracking, this might be important information.  Version numbers don’t provide any context for when an update occurred.

If a timestamp is useful, change your version property to be a java.util.Date or Calendar and use a <timestamp> (rather than using a <version>) element in the mapping file.

```
public class Product {
…
  private Date version;
…
}
<hibernate-mapping package=”com.intertech.domain”>
<class name=”Product”>
<id name=”id”>
<generator class=”native” />
</id>
<!–<version name=”version” type=”timestamp” generated=”always”/>–>
<timestamp name=”version” source=”db”/>
      <property name=”name” not-null=”true” />
<property name=”model” not-null=”true” />
<property name=”cost” type=”double”/>
</class>
</hibernate-mapping>
```

The <version type=timestamp”> and <timestamp> are equivalent, but it might help the reader to better see/understand that the version is an effectivity timestamp versus a numerical indicator.

You might also notice the source=”db” in the mapping configuration above.  The source attribute can be set to “vm” (for virtual machine) or “db” (for database).  When set to “db”, the database creates and manages the version timestamp (equivalent to generated=”always”) above.  As mentioned above, in the case of the database managing the version timestamp, your application may incur an additional hit of the database as Hibernate must first retrieve the

timestamp before updating the row.  Further, as the Hibernate documentation says:  “Not all `Dialects` are known to support the retrieval of the database’s current timestamp.”

## Optimistic Locking without a Version

Your application may need to deal with a legacy database and the database cannot be altered to contain a version (numeric or timestamp) column.  In this case, Hibernate provides two additional options for optimistic lock management.  You can use all the fields of the persistent object to check that nothing in a row was modified before updating a row.  Set the optimistic-lock=”all” attribute (and dynamic-update=”true”) in the <class> element to use this approach.  The SQL issued by Hibernate will then use each property as a check in the where clause as shown below.

> Hibernate:
> /* update
> com.intertech.domain.Product */ update
> Product
> set
> name=?
> where
> id=?
> and name=?
> and model=?
> and cost=?

comparison of the state of all fields in a row but without a version or timestamp property mapping, turn on `optimistic-lock="all"` in the `<class>` mapping

As a slight alternative to this approach, you can set optimistic-lock=”dirty” and then Hibernate will only use modified columns as part of the optimistic check (see below).  This last approach actually allows to transactions to update the same object concurrently legally, but only if they are modifying different columns – something that might be advantageous in a larger application setting where many people are accessing the same objects (thus rows) but rarely updating the same properties (columns).

> Hibernate:
> /* update

```
com.intertech.domain.Product */ update
Product
set
name=?
where
id=?
and name=?
```

See the Hibernate documentation for more details on these alternatives to versioning.

Come join us to learn more about Hibernate!  Sign up for Complete Hibernate at Intertech.com here.

## Recent Posts

The Future of Java: Recent Changes and What's Next

GitHub Acquisition, Useful Javascript, and More...

Case Study: Agile Implementation and Modernization of a Technology Stack

How Intertech Enabled a Renewable Energy Company to Launch Software Quickly

Russian Router Hack, Thinking Like a CEO, and More...

## Follow Us

## Categories

Categories

Select Category

SridharSrinivasan007 • 4 years ago

Hi Sir, I have tried "Optimistic Locking without a Version". It is not working as expected. I am going to explain what I did. I have student list page. each and every row edit link is there. When I click the edit link, the other page will open to update the student. Now i have opened the same student details in different window for updating.and I tries to update student one by one. If I use @Version then i am getting the StaleObjectStateException which we are expected Now I want to achieve the optimistic

getting the StaleObjectStateException which we are expected.Now I want to achieve the optimistic locking without version. but if I do that, I am not getting any exception instead the rows updated successfully. Do you have any idea why this is not working

Reply • Share ›

**Jim White** Mod ➚ SridharSrinivasan007 • 4 years ago

Sridhar, it could be a number of things. How are you designating the optimistic lock without version? Are you using all fields or dirty fields? Can you provide some indication (even a little code) of how you have this setup? the smallest mistake can cause the locking not to work. Also, are you sure the entity update is detected, the session is dirty and that it knows the object under concern needs updating? You can check a session for if it is dirty (isDirty method) after you change your entity.

Reply • Share ›

**Thilak Raj** • 4 years ago

I didnt give generated type as "always" so now in update statement, do i need to set version value manually with one increment to it???

Reply • Share ›

**subrat** • 5 years ago

Nice one!!! got clear idea about optimistic locking

Reply • Share ›

**Ramesh k swarnkar** • 5 years ago

Excellent Explanation about Optimistic Lock and Versioning. I appreciate it.
Will look for forward for more topics on Hibernate advance features.

Reply • Share ›

**Srinivas** • 5 years ago

Very Nice article!

Reply • Share ›

**Kannan** • 5 years ago

Excellent Post. Thanks.

Reply • Share ›

**Pramod** • 5 years ago

Nice one....

Reply • Share ›

**Robin** • 6 years ago

Great article.

Reply • Share ›

**Renato** • 6 years ago

Awesome post. Very clear! Thanks!

Reply • Share ›

**Chinmaya** • 6 years ago

Awesome. Nicely explained. Thanks

Reply • Share ›

**Prakash** • 6 years ago

Nice explanation. Thanks

Reply • Share ›

**desperado** • 6 years ago

Awesome! thanks for your post.
I&#39ve encountered a problem with hibernate versioning which couldn&#39t find any useful information from google, the scenarios as below:

EntityOne (identity = generated)
EntityTwo (identity = assigned)

session.saveOrUpdate(entityOne); << version always increased as expected for a detached object;
session.saveOrUpdate(entityTwo); << hibernate didn&#39t increase the version for a detached object;

The only different between two entity is identity generated strategy (assigned and generated)
Both entityOne and entityTwo save without changes! if any of the column revised, the version increased as expected.

Could anyone give a thought on this?

Reply • Share ›

**Dipankar Roy** • 6 years ago

Its really a nice post for beginer to understand the concept of Versioning . Thx a lot.

Reply • Share ›

**Mehul** • 6 years ago

Great post. Help me understand versioning and optimistic lock easily.

Reply • Share ›

**srinivas** • 6 years ago

Great Insight of Optimistic locking. Thanks for the good post.

Reply • Share ›

**sai** • 6 years ago

Great explanation, could just glance it in 5 mins.. Thanks a lot .

Reply • Share ›

**Sriram Krishnan** ➚ sai • 4 years ago

well written, easy to understand..

Reply • Share ›

## Recent Posts

- The Future of Java: Recent Changes and What's Next
- GitHub Acquisition, Useful Javascript, and More...

- Case Study: Agile Implementation and Modernization of a Technology Stack
- How Intertech Enabled a Renewable Energy Company to Launch Software Quickly
- Russian Router Hack, Thinking Like a CEO, and More...

Practical insights, tips, tutorials and examples from team of software development consultants and trainers.

## Contact Details

1575 Thomas Center Drive
Eagan, MN 55122
General: 651.288.7000

Training: 651-288-7109
Consulting: 651-288-7001
Toll Free: 800-866-9884

## About Intertech

Home
Consulting
IntertechU Course Selector