**Generator classes**

* Generator classes are used to generate the ‘**identifier’** for a persistent object. i.e. While saving an object into the database, the generator informs to the hibernate that, how the primary key value for the new record is going to generate
* Hibernate using different primary key generator algorithms, for each algorithm internally a java class is there for its implementation
* All Generator classes has to implement ‘**org.hibernate.id.IdentifierGenerator”** interface, And has to override **generate()** method. The logic to generate **‘identifier’** has to write in this method.
* In built-in generator classes, identifier generation logic has implemented by using JDBC.
* If we want we can write the user defined generator class and it should implement ‘**org.hibernate.id.IdentifierGenerator’** interface and has to override **generate()** method.
* To configure generator class we can use **<generator />** tag, which is the sub element of **<id/>** tag
* **<generator/>** tag has one attribute called “**class**” with which we can specify generator class-name.
* While configuring **<generator />** tag in mapping file, if we need to pass any parameters to generator class then we can use **<param />** tag, which is the sub element of **<generator />** tag.

**Example: HBM**

<hibernate-mapping>

<class … >

<id …. >

**<generator class="generator-class-name">**

**<param name="param-name">param-value </param>**

**</generator>**

</id>

</class>

</hibernate-mapping>

The following are the list of main generators we are using in the hibernate framework

1. sequence
2. assigned
3. increment
4. hilo
5. seqhilo
6. identity
7. native
8. uuid
9. guid
10. select
11. foregin

In the above generators list, are used for int,long,short types of primary keys, and uuid, guid are used when the primary key column type is String type (varchar2)

**1.) sequence (org.hibernate.id.SequenceGenerator)**

* This generator class is database dependent it means, we cannot use this generator class for all the database, we should know whether the database supports sequence or not before we are working with it
* Not has the support with MySql
* Here we write a sequence and it should be configured in HBM file and while persisting the object in the database sequence is going to generate the identifier and it will assign to the Id property of persistent object, Then it will store the persistent object into Database.

**NOTE:** MYSQL even won’t allow to create SEQUENCE Object in that database. Then where is the question of calling that from hibernate application?

**NOTE:** When we configured generator class for an entity, then we no need to assign identifier value to entity object while saving the entity. Even we assign the identifier value to entity, it will not consider our assigned value, it will use generator class generated value as identifier value. It applies to all generator classes.

**Steps to work with “sequence” generator**

**Step 1**: Create a sequence

**SQL> CREATE SEQUENCE ACCNO\_SEQ START WITH 1000 INCREMENT BY 1**

To get next value

**SQL> SELECT ACCNO\_SEQ.NEXTVAL FROM DUAL;**

To get current value

**SQL> SELECT ACCNO\_SEQ.CURRVAL FROM DUAL;**

**Step 2**: configure the sequence in hbm file.

**Account.hbm.xml**

<hibernate-mapping>

<class ……. >

<id name="accountId" column="ACNO">

<generator class="**sequence**" >

<paramname="sequence">**ACCNO\_SEQ**</param>

</generator>

</id>

<property ……/>

</class>

</hibernate-mapping>

**(OR)**

<id name=" accountId " column="ACNO">

<generator class=**"org.hibernate.id.SequenceGenerator"** >

<param name="sequence">**ACCNO\_SEQ** </param>

</generator>

</id>

**(OR)**

<id name=" accountId " column="ACCNO">

<generator class="sequence" />

</id>

**NOTE:** If we don’t configure any sequence name then it will take default sequence name**(HIBERNATE\_SEQUENCE** ). But hibernate won’t create the sequence, if already sequence is available with name “**HIBERNATE\_SEQUENCE**”, then it is used by hibernate. Otherwise hibernate raises the exception.

**NOTE:** But remember, if we enable **hbm2ddl.auto** property in hibernate configuration file, then hibernate will create the database objects if they are not exist.

**NOTE:** It is not advisable to use the default sequence, always prfer to create a different sequence to each enitity seperately.

**Step 3**: Create the entity and save it without assigning identifier.

1. **Session session = SessionUtil.getSession();**
2. **session.getTransaction().begin();**
3. **Account account = new Account();**
4. **account.setName("sekhar");**
5. **account.setBalance(5000);**
6. **Serializable id = session.save(account);**
7. **System.out.println("Account is created with accno : "+id);**
8. **session.getTransaction().commit();**

**NOTE**: When we execute the above code, we can find the sequence execution query on the console, which is used to get identifier for the saving enitity.

**Internal Code**

**public** **class** SequenceGenerator **implements** PersistentIdentifierGenerator, Configurable {

**public** **static** **final** String SEQUENCE = *"sequence";*

……………………………………………………………..

……………………………………………………………..

**public** **void** configure(. . .){

sequenceName = PropertiesHelper.getString(SEQUENCE, params,

**"hibernate\_sequence**"); // default sequence name

parameters = params.getProperty(PARAMETERS);

}

**public** Serializable generate(. . . ) {

PreparedStatement st =. . ..prepareSelectStatement(sql);

ResultSet rs = st.executeQuery();

rs.next();

Serializable result = . . iterate results…

**return** result;

}

}

**NOTE:** Now onwards for the following generator classes I just give the HBM configuration, you can use the same entity saving logic(which we used in the above example as part of step-3) to test them.

**2.) assigned(org.hibernate.id.Assigned)**

* This generator supports in all the databases
* This is the default generator class used by the hibernate, if we do not specify **<generator/>** element under **<id/>** element, then hibernate by default assumes it as “**assigned**” generator class.
* If generator class is assigned, then the programmer is responsible for assigning the identifier value to entity before saving into the database

**HBM:**

<id name="accountId" column="ACCNO">

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

</id>

(OR)

<id name=" accountId " column="ACCNO">

<generator class="**org.hibernate.id.Assigned**" />

</id>

(OR)

<id name=" accountId " column="ACCNO" />

**Internal Code:**

**public** **class** Assigned **implements** IdentifierGenerator, Configurable {

**public** Serializable generate(... ){

**final** Serializable id = . . . get the developer given id.

**if** (id==**null**) {

**throw** **new** IdentifierGenerationException(

"ids for this class must be manually assigned before calling save(): " + entityName);

}

**return** id;

}

}

**NOTE**: While testing don’t assigne identifier value to entity and then try to save entity then it throws **IdentifierGenerationException.**

**NOTE:** While testing with assigned generator class, identifier type in enity should be some object type rather than primitive type.

**3.)increment(.hibernate.id.IncrementGenerator)**

* This generator supports in all the databases, so it is database independent generator class.
* This generator is used for generating the id value for the new record by using the formula

**Max of id value in Database + 1**

* If there is no record initially in the database, then for the first time this will saves primary key value as 1.

**HBM:**

<id name="accountId" column="ACCNO">

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

</id>

(OR)

<id name=" accountId " column="ACCNO">

<generator class=" **org.hibernate.id.IncrementGenerator** " />

</id>

**Internal Code:**

**public** **class** IncrementGenerator **implements** IdentifierGenerator, Configurable {

**public** **synchronized** Serializable generate(. . .){

**if** (sql!=**null**) {

getNext( session );

}

**return** *... final number after adding ‘1’*

}

**public** **void** configure(. . .) {

. . . . . .

. . . . . .

sql = "select max(" + column + ") from " + buf.toString();

}

**private** **void** getNext( SessionImplementor session ) {

PreparedStatement st = ....prepareSelectStatement(sql);

ResultSet rs = st.executeQuery();

**if** ( rs.next() ) {

next = rs.getLong(1) + 1;

}

}

}

**The HiLo Algorithm**

The HiLo (High/Low) algorithm knows how to generate unique number series using two values: the high and the low. The high value is used as a base for a series (or range) of numbers, while the size of this series is donated by the low value. A unique series is generated using the following steps:

1. Load the and atomically increment the high value
2. Multiple the high value by the low value (max\*low), the result is the first number (lower bound) of the current series
3. The last number (higher bound) of the current series is donated by the following calculation: (max\*low)+low-1
4. When a client needs to obtain a number the next one from the current is used, once the entire series has been exhausted the algorithm goes back to step 1

**Example:** suppose that the current high value in the database is 52 and the low value is configured to be 32,767. When the algorithm starts is loads the high value from the database and increments it in the same transaction (the new high value in the database is now 53). The range of the current numbers series can now be calculated:

• Lower bound = 52\*32767 = 1,703,884

• Upper bounds = 1,703,884+32,767-1 = 1,736,650

All of the numbers in the range of 1,703,884 to 1,736,650 can be safely allocated to clients, once this keys pool has been exhausted the algorithm needs to access the database again to allocate a new keys pool. This time the high value is 53 (immediately incremented to 54) and the keys range is:

• Lower bound = 53\*32,767 = 1,736,651

• Upper bounds = 1,736,651+32,767-1 = 1,769,417

And so on

The big advantage of this algorithm is keys preallocation which can dramatically improve performance. Based on the low value we can control the database hit ratio. As illustrated using the 32,767 we hit the database only once in a 32,767 generated keys. The downside (at least by some people - but in my opinion this is a none-issue) is that each time the algorithm restarts it leaves a 'hole' in the keys sequence.

Hibernate has several HiLo based generators: TableHiLoGenerator, MultipleHiLoPerTableGenerator, SequenceHiLoGenerator

**TableHiLoGenerator**

A simple HiLo generator, uses a table to store the HiLo high value. The generator accepts the following parameters

* **table** - the table name, defaults to **'hibernate\_unique\_key**'
* **column** - the name of the column to store the next high value, defaults to **'next\_hi'**
* **max\_low** - the low number (the range) defaults to **32,767** (Short.MAX\_VALUE)

**MultipleHiLoPerTableGenerator**

A table HiLo generator which can store multiple key sets (multiple high values each for a different entity). This is useful when we need each entity (or some of the entities) has its own keys range. It supports the following parameters:

* **table** - the table name, default to 'hibernate\_sequences'
* **primary\_key\_column** - key column name, defaults to 'sequence\_name'
* **value\_column** - the name of the column to store the next high value, defaults to 'sequence\_next\_hi\_value'
* **primary\_key\_value** - key value for the current entity (or current keys set), default to the entity's primary table name
* **primary\_key\_length** - length of the key column in DB represented as a varchar, defaults to 255
* **max\_low** - the low numer (the range) defaults to 32,767 (Short.MAX\_VALUE)

The generator uses a single table to store multiple high values (multiple series), when having multiple entities using the same generator Hibernate matches an entity to a high value using the primary\_key\_value which is usually the entity name. A sample table can look like

|  |  |
| --- | --- |
| sequence\_name | sequence\_next\_high\_value |
| ENTITY1 | 234 |
| ENTITY2 | 876 |
| ENTITY3 | 8 |

**SequenceHiLoGenerator**

A simple HiLo generator but instead of a table uses a sequence as the high value provider.

* **sequence** - the sequence name, defaults to 'hibernate\_sequence'
* **max\_low** - the low number (the range) defaults to 9.

**4.) hilo( org.hibernate.id.TableHiLoGenerator)**

* This generator is database independent
* hilo uses a hi/lo alogorithm to generate identifiers.
* Hilo algorithm generate identifiers based on the given table and column(stores high value). Default table is ‘**hibernate\_unique\_key’** column is ‘**next\_hi**’.

**HBM:**

<id name="accountId" column="ACCNO">

<generator class="**hilo**" >

<param name="table">HIGH\_VAL\_TAB</param>

<param name="column">HIGH\_VAL\_COL</param>

<param name="max\_lo">60</param>

</generator>

</id>

(OR)

<id name="accountId" column="ACCNO">

<generator class=" **org.hibernate.id.TableHiLoGenerator**" >

<param name="table">HIGH\_VAL\_TAB</param>

<param name="column">HIGH\_VAL\_COL</param>

<param name="max\_lo">60</param>

</generator>

</id>

**Internal code**

class TableHiLoGenerator extends TableGenerator {

………………..

}

class TableGenerator {

**public** **static** **final** String COLUMN = "column";

**public** **static** **final** String DEFAULT\_COLUMN\_NAME = "next\_hi";

**public** **static** **final** String TABLE = "table”;

**public** **static** **final** String DEFAULT\_TABLE\_NAME = "hibernate\_unique\_key";

……………………………………….

}

**5.) seqhilo(org.hibernate.id.SequenceHiLoGenerator)**

* It is just like hilo generator class, But **hilo** generator stores its high value in table, where as **seqhilo** generator stors its high value in sequence.

**HBM:**

<id name="accountId" column="ACNO">

<generator class="**seqhilo**" >

<param name="sequence">ACCNO\_SEQ</param>

<param name="max\_lo">5</param>

</generator>

</id>

(OR)

<id name=" accountId " column="ACCNO">

<generator class="seqhilo" >

<param name**="org.hibernate.id.SequenceHiLoGenerator "**>ACCNO\_SEQ</param>

<param name="max\_lo">5</param>

</generator>

</id>

**Q.) How to work with MYSQL database?**

* install the MYSQL database.
* Select All Programs 🡪 MYSQL 🡪 MYSQL 5.0 🡪 mysql command line client
* Give password
* In the MYSQL prompt give the following commands
  + **create database mydb**
  + **use mydb**
* Now perform database operations normally.
* When we login to database next time onwards don’t give **create database** command, just give **use** command.

**6.) identity(org.hibernate.id.IdentityGenerator)**

* This is database dependent, actually it’s not working in oracle.
* Identity columns are support by DB2, MYSQL, SQL SERVER, SYBASE and HYPERSYNCSQL databases.
* This identity generator doesn’t needs any parameters to pass

**Syntax to create identity columns in MYSQL database:**

CREATE TABLE STUDENT(

SNO INT(10) NOT NULL AUTO\_INCREMENT,

COURSE CHAR(20),

FEE FLOAT,

NAME CHAR(20),

PRIMARY KEY (SNO)

)

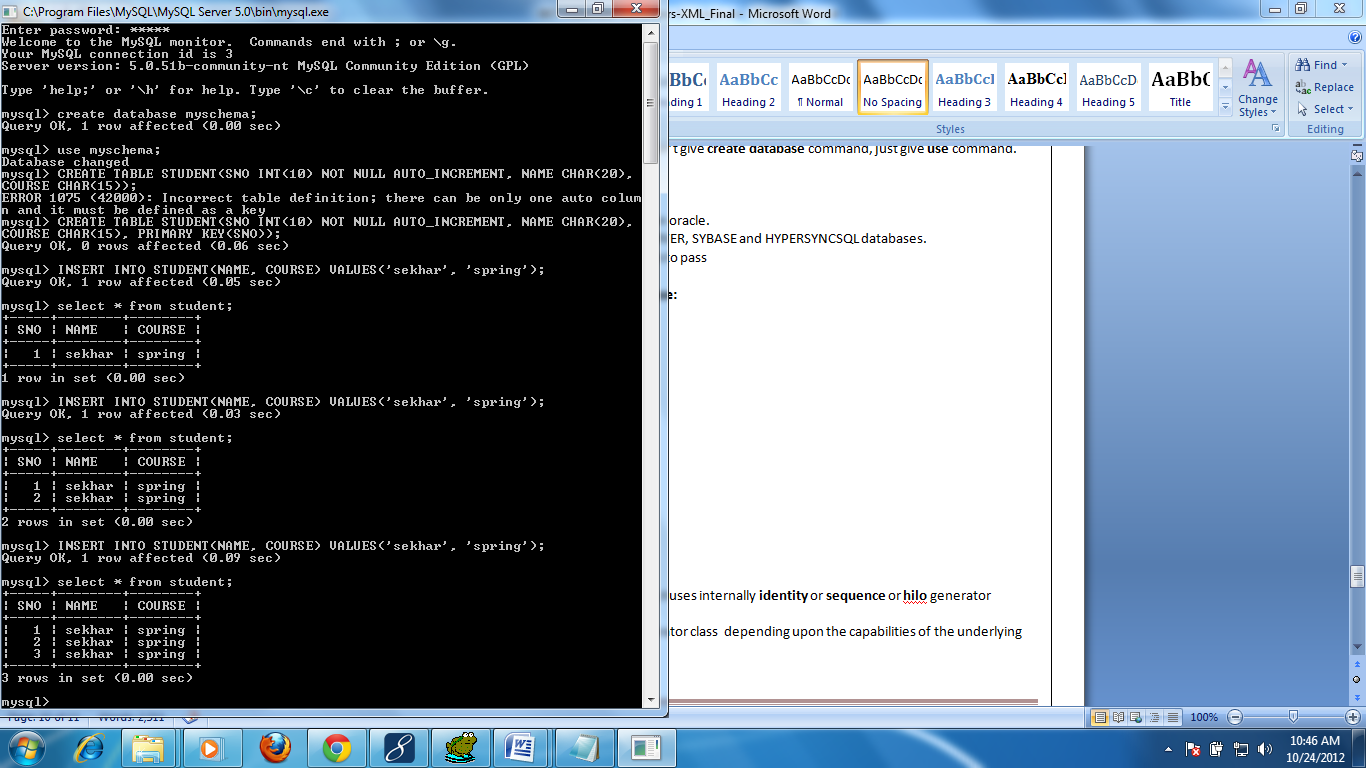
**HBM:**

<id name=" accountId " column="ACNO">

<generator class="identity" />

</id>

**Example**: To create identity columns in MYSQL database.



**7.) native**

* **native** is not having any generator class because, it uses internally **identity** or **sequence** or **hilo** generator classes.
* **native** picks up **identity** or **sequence** or **hilo** generator class depending upon the capabilities of the underlying database.

**HBM:**

<id name="accountId" column="ACNO">

<generator class="**native**" >

<param name=”sequence”>ACCNO\_SEQ</param>

<param name="table">HIGH\_VAL\_TAB</param>

<param name="column">HIGH\_VAL\_COL</param>

<param name="max\_lo">60</param>

</generator>

</id>

**NOTE**: test with mysql and oracle

* If we connect to mysql it takes identity.
* If we connect to oracle it takes sequence.

**8.) uuid(org.hibernate.id.UUIDHexGenerator)**

* **uuid** uses a 128-bit uuid algorithm to generate identifiers of type string.
* **uuid** generated identifier is unique with in a network.
* **uuid** algorithm generates identifier using IP address.
* **uuid** algorithm encodes identifier as a string(hexadecimal digits) of length 32.
* Generally uuid is used to generate passwords.

**HBM:**

<id name="accountId" column="ACNO">

<generator class="uuid" >

</generator>

</id>

(OR)

<id name=" accountId " column="ACCNO">

<generator class="org.hibernate.id.UUIDHexGenerator " >

</generator>

</id>

**select:**  select retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.

**guid:**  uses a database generated guid string on MS-SQL and MYSQL.

**foreign:**  foreign uses the identifier of another associated object. Usually uses in conjunction with a <one-to-one> primary key association.

**User Defined Generator class**

* When we feel the existing generator classes are not fit for our requirement, then we will go for user defined generator class.

**Steps to implement user defined generator class.**

**Step 1:** Take any java class and implement **org.hibernate.id .IdentifierGenerator** and override **generate()** method. In this method implement the identifier generation logic as per the requirement.

**Step2:** Give the generator class entry in the **<generator>** tag of HBM file.

**Step 1: AccountNumberGenerator.java**

1. **package com.sekharit.hibernate.id;**
2. **import java.io.Serializable;**
3. **import java.sql.Connection;**
4. **import java.sql.ResultSet;**
5. **import java.sql.Statement;**
6. **import org.hibernate.HibernateException;**
7. **import org.hibernate.engine.SessionImplementor;**
8. **import org.hibernate.id.IdentifierGenerator;**
9. **public class AccountNumberGenerator implements IdentifierGenerator {**
10. **public Serializable generate(SessionImplementor session, Object object)**
11. **throws HibernateException {**
12. **String accno = "SBH";**
13. **try {**
14. **Connection connection = session.connection();**
15. **String query = "SELECT TO\_CHAR("**
16. **+ "ACCNO\_SEQ.NEXTVAL,'FM00000000') FROM DUAL";**
17. **Statement statement = connection.createStatement();**
18. **ResultSet resultSet = statement.executeQuery(query);**
19. **if (resultSet.next()) {**
20. **accno = accno + resultSet.getString(1);**
21. **}**
22. **} catch (Exception e) {**
23. **e.printStackTrace();**
24. **}**
25. **return accno;**
26. **}**
27. **}**

**Step 2:HBM**

<id name="accountId" column="ACCNO">

<generator class="com.sekharit.hibernate.id.AccountNumberGenerator" />

</id>

**NOTE**: In the above user defined generate class I used database sequence to generate next value… And I got the connection from session object, but this method is deprecated in latest versiton. So we need to physically create the connection in the latest versions.