1. **What is hibernate filter ?**

Hibernate 3 provides an innovative approach to handling the data with visibility rules. Hibernate filter is a global , named ,parameterized filter that can be enabled or disabled for the hibernate session. It allows hiberate to predefine filter criteria and apply that filter to class level or collection level. It is used to form a restriction clause similar to where clause. It is an alternative approach to DB views , where clause and Criteria API. It is similar to DB Views executed in application level and its parameterized. Filter can be defined by using the XML or Annotations. The application can decide at runtime which filter needs to be enabled and what is the value to be applied.

Filter definition has name and array of parameter. The parameter has name and type.

1. **Using the Filter in XML :**

The filter needs to be defined in hibernate mapping tag and can eb applied to class or collection.

Ex:

The following filter , filter the data based on the given date.

<hibernte-mapping>

**// filter definition**

<filter-def name=”stockRecordFilter”>

<filter-parameter name=”stockRecordFilterParam” type=”date” />

</filter-def>

<class name=”com.stock” table=”stock” >

<set name=”stockDailyRecords” table=”stock\_daily\_record”>

<key column=”STOCK\_ID” not-null=”true” />

<one-to-many class=”com.StockDailyRecord” />

**//apply defined filter to the colletion**

<filter name=” stockRecordFilter” condition=”stDate>=: stockRecordFilterParam” />

</set>

</class>

<hibernte-mapping>

Condition =”stDate>=: stockRecordFilterParam” , stDate is the proeprties from StockDailyRecord , stockRecordFilterParam is an filter parameter value to be set at runtime by an application.

1. **Using the Filter inAnnotations :**

The filter definition should be define by using the @FilterDef from org.hibernate.annotations.FilterDef or org.hibernate.annotations.FilterDefs. The filter param should be defined as @ParamDef.

Ex:

@Entity

@FilterDef(name=”stockRecordFilter” parameters=@ParamDef(name=”stockDailyFilterParam” type=”date”)

@Table(name=”stock”)

Public class Stock implements java.io.Serializable{

@one-to-many(fetch=FetchType.LAZY,mappedBy=”stock”)

@Filter(name=” stockRecordFilter” condition=”stDate>=:stockRecordFilterParam”)

Set<StockDailyRecord> stockDailyRecords;

Public Set<StockDailyRecord> getStockDailyRecords {

Return this.stockDailyRecords;

}

}

Code to use filter using the hibernate session,

Filter can be enabled and disabled by using the enableFilter() and disableFilter() methods available in hibernate session.

Session session=sessionFactory.getSession();

// Enable the session

Session.enableFilter(“stockRecordFilter”).setParameter(“stockRecordFilterParam”,”some value”); // which returns the filter Instance

Stock stock=session.get(Stock.class,2);

//this set has the filter applied

Set <StockDailyRecords> sets=stock.getStockDailyRecords();

Session.diableFilter(“stockRecordfilter”)

//clean up the session

//retrieve the stock records now

Stock stock=session.get(Stock.class,2);

//this set doent have the filter applied

Set <StockDailyRecords> sets=stock.getStockDailyRecords();

In the above example, the same filter can be applied to class level also.

Another Ex:

@FilterDef(name="studentFilter", parameters={

@ParamDef( name="maxAge", type="integer" ),

@ParamDef( name="minAge", type="integer" ),

@ParamDef( name="minNumber", type="integer")

})

@Filters( {

@Filter(name="studentFilter", condition=":minAge <= age and :maxAge >= age"),

@Filter(name="studentFilter", condition=":minNumber <= number")

} )

In this above ex, there are 2 conditions are applied to same filter named. When use more than one @filters use the @Filters as like above.

**Using the filters in the jointable:**

1. **What is query caching in hibernate?**

Query result can also be cached in hibernate. This is useful for the queries that are frequently validated with same parameters. Query caching should be used with secone level cache.

Query cahcing is not enabled by default.

To enable the query caching we have to do the following change in config file,

Hibernate.cache.use\_query\_cache=true;

This setting creates two cache regions,

1. Org.hibernate.cache.internal.StandardQueryCache – cache the query result
2. Org.hibernate.cache.spi.UpdateTimeStampCache – hold the timestamp for the most recent updates to queryable tables.

To use the query caching , set org.hibernate.Query.setCahable(true), This will look for the result in cache , if not find execute the query on database and add the results to the cache. The query cache regions can be named by a specific name.It can be achieved by setting the org.hibernate.Query.setRegion(“regionname”. This allows the query caching to referesh only the particular region of the cache. To refresh the Query cache region use org.hibernate.Query.setCacheMode(CachMode.REFRESH).

**Ex:**

List list=session.createQuery(“from employee where empno>=?)

.setInt(0,45);

.setCachable(true)

.setRegion(“emplist”)

.setCacheMode(CacheMode.REFRESH)

.list();

**How its represented inside the query cache:**

The combination of query and the parameters passed to that query used as a key , and the identifiers of the result used as value.

[“from employee where empno>=?,[45]] - > [2,4]]

2 and 4 – are an identifier of the employee entity , by using this id it will load the actual entity from the cache.

1. **How hibernate second level cache works?**

Caching is the facility provided by the ORM framework ,which speeds up the web applications by reducing the number of database hits.

**Hibernate supports 2 types of caching:**

* **First level :** Associated with session objects. Its enabled by default.
* **Second Level :** Associated with session factory object and its global to application. And its not enabled by default. It has to be manually enabled in hibernate properties.

There are 4 different second level cache providers available from different vendors,

^ EHCache (Easy Hibernate Cache) from hibernate framework

^ OSCache from Open Symphony

^ SwarmCache

^ JBoss TreeCache

There are 4 different cache usage available,

1. Read-only : cache will work only for read operation
2. Nonstrict-read-write : cache work for read and write but one at a time
3. Read-write : cache work for read and write simultaneously
4. Transaction : cache work for transaction

EHCache is the cache mostly used in the hibernate frame work which support first 3 usage and not act as transaction cache.

**Enable Second Level Cache: (4 steps)**

1. Add the cache provider in hibernate properties section in the config file.

<properties name=”hibernate.cache.provider\_class” > org.hibernate.cache.EhCacheProvider

</property>

1. Set the second level cache property as true

<property name=”hibernate.cache.use\_second\_level\_cache” > true </property>

1. Specify the cache element for the class in hbm file with appropriate value after the class element.

<cache usage =”read-only” />

1. Create ehcache.xml file

Ex:

<ehcache>

<defaultCache

maxElementsInMemory=”100”

eternal=”false”

timeToIdealSeconds=”120”

timeToLiveSeconds=”200”

>

</defaultCache>

<cache name=”com.Employee”

maxElementsInMemory=”100”

eternal=”false”

timeToIdealSeconds=”120”

timeToLiveSeconds=”200”

/>

</ehcache>

From the above file default cache is used for all persistent classes. We can also define the cache for the specific persistent class explicitly.

If we specify the eternal=”true” , we no need to specify the ideal and live seconds, hibernate itself handle this.

timeToIdealSeconds – defines how many seconds object can be ideal in the second level cache

timeToLiveSeconds – defined how many seconds object can be at second level cache whether its ideal or not.

**Second Level Cache Working Strategies:**

1. When the first session try to load an entity at very first time , it will not be available in first level(session) cache and second level cache. So it hit the database and load an entity to first level cache also to second level cache.
2. Second time session try to load the same entity , it look for the first level cache for the cached entity , if its available then load the entity
3. If the cached entity is not available on the first level cache then it look for the second level cache for the cached entity.
4. If the entity is available in the second level cache ,its copied to first level cache and loaded
5. Now, second session comes in try to load the entity , it lookfor the first levelcache (it will not be available) , then look for the second level cache(available in session factory) , so load the entity into first level cache and return the result.
6. Read and write operation on the cached entity is control by the caching mode specified in the class mapping of the each entity.
7. **How to use the hibernate second level ehcache in clustered environment ?**

**http://www.surekhatech.com/blog/hibernate-second-level-cache-using-ehcache-for-cluster-environment**

[**http://www.ehcache.org/documentation/2.8/replication/**](http://www.ehcache.org/documentation/2.8/replication/)

1. **What is hibernate interceptor ?**

It is an hibernate feature which allows application to intercept on some hibernate events to add additional functionality or implement some generic functionality. Each method on the hibernate session has associated events. The interceptor interface provides call back methods from session to application. The interceptor are created by implementing the Interceptor Interface or extending the EmptyInterceptor class. The interceptor feature is very useful to implement the Audit log functionality.

There are two types of interceptor , (interceptors are enabled by using any of these ways)

1. Session Factory Interceptor – created by setting the interceptor instance as into Configuration object.

New Configuration().setInterceptor(new Interceptor());

1. Session Interceptor – created by passing the interceptor instance into the openSession().

Session=sessionFactory.openSession(new Interceptor());

**Methods available in Interceptor:**

onSave() – called when the object is saved, the object is not saved into database yet.

onFlushDirty() –called when the object is updated, the object is not updated into database yet

preFlush() – called before saved, updated or deleted object are committed into database

postFlush()- called after saved , updated or deleted objects are committed into database

onDelete() – called when object is deleted, objects are not delete from database yet.

**Implementation Details:**

<http://www.mkyong.com/hibernate/hibernate-interceptor-example-audit-log/>

1. **What is hibernate envers?**

Envers is an library added as part of core module. Envers provides an easy way of auditing the persistent class / entities in hibernate. It helps to easily achieve the audit functionality. It has built in mechanism to maintain the history tables for the audited entity. It is similar to Version Control System. It maintains the versioning / revisions for each transaction unless it didn’t modify the audited properties.

In previsions version of hibernate we need to register the listener. But from 3.5 if we add the envers jar file in class path it automatically register the listeners.

**Enver Features:**

* Maintain the historical data
* Auditing the mapping defined in the hibernate mapping file
* Querying the historical data

**How Enver works?**

* Add the envers jar and annotations jar in classpath
* Add the @Audited annotation in the entity or some of the properties in the entity
* Create the audit table manually or by setting the below property in hibernate configuration file as

<property name=”hbm2ddl” > update or create or create or drop <property>

The audit table is created in the form of entityname\_aud

* The @Audited can be applied to entity or property
* The history data can be fetched by using the AuditReader API which has AuditReader interface and AuditReaderFactor
* If the @Audited maintained at class level , all properties chages are record in the audit table.
* In the @Audited entity , if some of the property doesn’t want to involve in the auditing , then use the @NotAudited annotations then these properties are not stored in the audit table.
* If the @Audited is not used in entity level but maintained at property level , then only that properties changes are recorded in the audit table
* The audit table contains the two extra columns other than defined in the persistent entity.
* The two extra columns are , REV and REVTYPE

REV – Revision number which maintains the versioning of the entity

REVTYPE – Revision type which specifies that the action taken on that entity , which has the possible below values

0 – creation

1 – modification

2 – deletion

By using this version number we can fetch all the entities which has the same version number and timestamp

* There is one more table create as part of auditing is REVINFO , which has two fields

REV – Revision number

REVTSTMP

**Using the AuditReader API:**

Audit Reader API is used to fetch the historical data of the particular entity and fetch the particular version of the entity.

**Fetch the historical data:**

AuditReader reader=AuditRefaderFactory.get(sessionFactory.openSession())

List<Number> revisions=reader.getRevisions(person.class, new Integer(2));

For(Number number:revisions){

}

**Fetch the previous version/Revision of the entity:**

Person person=(Person) reader.find(Person.class, new Interger(1),2);

1. **What is multi-tenancy in hibernate ?**

Multitenancy is an technique used in software applications where the single application services for the multiple client. These clients are called as tenants.

The challenge here is to maintain the isolation of data between the clients. The client should have access to only his data and they should have access to the other client’s information.

There are 3 approaches are available to achieve this muti-tenancy

1. **Separate Database:**

Separate database is maintained to each tenant .so the data integrity is maintained. The solution in this approach is , maintain the separate connection pool for each tenant and the pool is chosen using the tenants identifier based on the user logged into the application.

1. **Separate Schema:**

In this approach , separate schema is used on single database for each of the tenant.

There are 2 solutions available in this approach,

1. Separate connection pool is used for each schema and the pool is chosen based on the tenant identifier.
2. The application point to default schema of the database and the different schema is selected using the SET SCHEMA SQL statement. Single connection pool is used and before getting the connection tenant identifier is used to alter the schema.
3. **Partitioned Data:**

In this approach, single database and single schema is used. All tenant data stored in the single database schema and the data are isolated based on the discriminator value which may a single column to complex SQL query.

The solution is to maintain the single connection pool and every SQL query is altered to reference the discriminator value of the tenant.

**How to achieve multi-tenancy in hibernate:**

To start the multitenant connection in hibernate , we use

Session session=**sessionFactory.withOptions().tenantIdentifier(“identifier”).openSession();**

The multitenant approach should be specified in the hibernate.cfg.hbm.xml.

The multitenant approach are as follows,

NONE :- default approach . it doesn’t take the tenant identifier. if we specify , it throws an exception

DATABASE:

SCHEMA

DISCRIMINATOR :- which should be implememted in the hibernate 5.0 version.

The multitenant approach other than NONE, should pass the tenant identifier to open the session , if we are not passed the tenant identifier it throws an exception.

To use the DATABASE and SCHEMA approach , we need to specify the multitenant connection provider.

There are 3 ways to specify the multitenant connection provider.

1. Use the **hibernate.multi\_tenant\_connection\_provider** settings in hibernate.cfg.xml file. Specify the multitenant connection provider implementation class as the value.
2. Directly pass the multitenant connection provider implementation class to org.hiberntae.boot.registry.StandardServiceRegistryBuilder

Steps to create the multitenant hibernate application:

1. Create the connection provider class which implements the MultiTenantConnectionProvider interface – This is responsible for providing the connections configured with the right schema specified
2. Create the Schema resolver class by implementing the CurrentTenantIdentifierResolver – This is responsible for resolving the name of the schema used.
3. Specify these classes and the multitenant approach in the hibernate.cfg.xml file

**<property name="hibernate.multiTenancy" value="SCHEMA"/>**

**<property name="hibernate.tenant\_identifier\_resolver" value="com.toptal.andrehil.mt.hibernate.SchemaResolver"/>**

**<property name="hibernate.multi\_tenant\_connection\_provider" value="com.toptal.andrehil.mt.hibernate.MultiTenantProvider"/>**

Ref Link :

**http://ganeshghag.blogspot.in/2014/09/database-multi-tenancy-using-hibernate-4.html**

**Hibernate 5 Features:**

Hibernate 5 introduce the following features.

1)Bootstrap:

2) Modularity

3) Bytecode augumentation

4) AttributeConverter

5)Hibernate search

6) OGM

7) Envers

**8) Java 8 support**

Hibernate 5 supports the java 8 new date time API. The java 8 date time classes can be used as the basic types . Hibernate 5 automatically maps it to the data base sql date type. Before hibernate 5, this requires the explicit attribute converter.

Ex:

@Entity

Public class Book{

Private Long id;

Private LocalDate publishingDate;

}

1. **Stream() method on Query Interface:**

Hibernate 5.2 allows to process the Query result as java 8 stream. It introduced the stream() method on Query Interface. It is the implementation of the ScrollableResultSet. When the result set is too big , stream() is an appropriate way to process the record one by one.

Ex:

Stream<Object[]> books = session.createNativeQuery("SELECT b.title, b.publishingDate FROM book b").stream();

books.map(b -> new BookValue((String)b[0], (Date)b[1]))

.map(b -> b.getTitle() + " was published on " + b.getPublishingDate())

.forEach(m -> log.info(m));

1. **Fetch Multiple entities by their primary key:**

The new MultiIdentifierLoadAccess interface provides an efficient way to fetch the multiple entities using their primary key in a single query. We just need to call the byMultipleIds() method in the session to get an instance of MultiIdentifierLoadAccess. Then pass the primary key values into the multiLoad() of MultiIdentifierLoadAccess instance.

MultiIdentifierLoadAccess<Book> multi=session.byMultipleIds(Book.class);

List<Book> books=multi.multiLoad(1L,2L,3L);

It creates the query as where book\_id in(?,?,?).

**Hibernate Architecture:**

Application

Persistent Objects

Hibernate

Hibernate mapping

Hibernate.properties

Database

* No arg constructor is required for all the persistent classes.
* Hibernate creates objects using reflection

**Hibernate mapping file:**

* The type attribute mentioned in the hibernate mapping file is not java data type or sql datatypes and this is called Hibernate Mapping Types, which are acts as converter to convert the java to SQL data types and vice versa.

**Hibernate configuration:**

* Hibernate has its own connection pool. It also supports third party connection pools to use.
* Hibernate connects to the data base on behalf of the application. For that it needs to get the connection.

The hibernate provided build in connection pool is used as like below,

<hibernate-configuration>

<session-factory>

<property name=”connection.pool\_size”>1</property>

</session-factory>

</hibernate- configuration >

* Sessionfactory is the global factory responsible for the particular database. If we use multiple databases , we have to use multiple session factories.

**Hibernate Startup:**

* Hibernate will be started by building the sessionfactory object.
* Sessionfactory is the thread safe object and it is initialized only once.
* Once hibernate startup is done , we need to configure the logging system.
* getCurrrentSession() :- return the current session associated with the thread if one exists. If no session is associated with the thread , it creates the sessions and bound to the thread. Always better to use the getCurrentSession() method .
* The session is unbounded from thread , when the transaction commits or rollback.
* The best way to use the session is session-per-operation.
* Set is the most commonly used collection mapping
* Without calling the save() or update() method ,Hibernate automatically detects the changes in the persistent objects and collection and update these changes at end of the unit of work. This is called automatic dirty checking. The end of the unit of work is called flushing. The flushing happens at the time of commit() or rollback() of the database transactions.
* The changes can even happens when the object is in detached state. And it will be update to the database when it is associated with the session.
* Collection of values can be associated with hibernate entity aslike

<set name=”emails” table=”PERSON\_EMAIL”>

<key column==”PERSON\_ID” />

<element type=”sstring” column=”EMAIL\_ADDR” />

</set>

* Always use the “inverse=true” one-to-one or one-to-many collection mappings. It means which side is responsible to take care of the relationship. Many-to-one doesn’t have the inverse-true.
* There are two table STOCK and STOCK\_DAILY\_RECORDS and they have one-to-many relationship. The STOCK has set of STOCK\_DAILY\_RECORDS . If the inverse =true is specified in the set for STOCK\_DAILY\_RECORDS then I is the relationship owner . When we update the STOCK , it wont update the STOCK\_DAILY\_RECORD.

If the inverse=false is used in the set, then STOCK is the owner of the relationship and is responsible for updating the STOCK\_DAILY\_RECORD. By default inverse=false.

**Locking in Hibernate: (What is hibernate locking ?)**

Locking refers to the action that prevents data from changing between the time it is read and the time it is updated.

There are different locking strategies available as,

1. Optimistic Locking
2. Pessimistic Locking

Hibernate provides mechanism for implementing these locking starategies.

**Optimistic Locking:**

Optimistic Locking assumes that the concurrent transaction doent not conflict with other , so it doesn’t required to lock the database resources.

This can be achieved by maintaining the version number , so that when the transaction updating the data it can make sure that no other transaction modifies its data. If its modified by other transaction, then the current transaction roll back the changes .

There are two mechanism available to maintaining the versioning of data,

1. Version number
2. Timestamp – less reliable way of maintaining the optimistic locking

Both can be achieved by using the @Version in the versioning column. The versioning column can be of any type. Usually it will be Integer. If the versioning column is Date type , then it update the timestamp.

The version number can be database generated version number generated by trigger.

It should be mentioned in the column with the annotation as @Generated(GenerationTime.always)

Ex: version number

@Entity

**public** **class** Flight **implements** Serializable {

...

    @Version

    @Column(name="OPTLOCK")

    **public** Integer getVersion() { ... }

}

In HBM file,

<version

        column="version\_column"

        name="propertyName"

        type="typename"

        access="field|property|ClassName"

        unsaved-value="null|negative|undefined"

        generated="never|always"

        insert="true|false"

        node="element-name|@attribute-name|element/@attribute|."

/>

Ex: Timestamp

The timestamp value can be fetched from database or VM . This should be mentioned as @Source(SourceType.DB) or @Source(SourceType.VM).

@Entity

public class Flight implements Serializable {

...

    @Version

    public Date getLastUpdate() { ... }

}

In hbm file,

<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|."

/>

**Optimistic Locking Scnerio:**

1. User 1 loads DomesticCat#1066 without upgrade lock.
2. User 2 loads DomesticCat#1066 without upgrade lock.
3. User 2 changes cat's name and commits.
4. User 1 changes cat's birthday, attempts to commit, an exception is thrown.

**Pessimistic Locking:**

Pessimistic locking assumes that concurrent transactions are conflict with each other so that data base resource should be locked until they finish using the resource(row).

In hibernate , just specify the isolation level to obtain the JDBC connection , and let the database handles the locking

Hibernate always uses locking mechanism of the database and never lock the object in memory.

**LockMode Class:**

The LockMode class is used to obtain the different type of locks in hibernate.

**LockModes:**

**LockMode.READ :** obtain the lock when its read data.

**LockMode.WRITE:** Obtain the lock when hibernate inserts or updated the data.

**LockMode.NONE:** No lock is used.

**LockMode.UPGRADE:** obtain the lock and doesn’t not allow other transaction to update the data until it releases it. Upgrade means that we want to modify the loaded object and don’t want other process to change the data until we process the data.

**Scenerio:**

1. User 1 loads DomesticCat#1066 *with* upgrade lock.
2. User 2 loads DomesticCat#1066 without upgrade lock.
3. User 2 changes cat's name and attempts to commit. This operation is not allowed until User 1 releases the lock, so the database blocks and User 2's session sits and waits.
4. User 1 changes cat's birthday, commits.

User 2's update can now try to commit, but since they used optimistic locking now they will be the one who sees an exception.

**LockMode.UPGRADE\_NOWAIT:**

The above mentioned lock modes can be specified in any one of the following calls

1. Session.load()
2. Session.lock()
3. Query.setLockMode()

**Diffrerence between optimistic lock and pessimistic lock:**

|  |  |
| --- | --- |
| **Optismistic** | **Pessimistic** |
| Object is not locked instead object state is saved | Object is locked |
|  |  |

**What are the different lock modes ?**

**Exclusive:**

Exclusive lock prevents the other users read the its locked resource. It is called as write locks. When one transaction holds an exclusive lock on data ,it prevents other transactions access the same data. The lock is on hold until the transaction release the lock by issuing the rollback or commit. It allows only one transaction allows to update the data and no other transaction can get the any type of the lock on the data.

The lock is with INSERT, DELETE, UPDATE

**Shared:**

Shared lock allows the other users to read the locked resources but they cannot update it. It is called read locks. When one transaction holds an shared lock on data , when the second transaction request the same data can be given the shared lock. Both transactions has the read access on the same data. Updating the data is not allowed until the shared lock is released. None of the transactions are allowed to update the data.

The lock is with SELECT

**Update:**

This is similar to exclusive lock. The difference is when one transaction can acquire the update lock when the other transaction already have an shared lock on the same data. Now the data has 2 locks as update lock by second transaction and shared lock by first transaction. Once the second transaction update the data , the update lock is converted to exclusive lock.

The lock is with SELECT..FOR UPDATE