**Hibernate Cache Mechanism:**

Caching is a mechanism for storing the loaded objects into a cache memory.  The advantage of cache mechanism is, whenever again we want to load the same object from the database then instead of hitting the database once again, it loads from the local cache memory only, so that the no. of round trips between an application and a database server got decreased.  It means caching mechanism increases the performance of the application

In hibernate we have two levels of caching

* First Level Cache [ or ] Session Cache

Second Level Cache [ or ] Session Factory Cache [ or  ] JVM Level Cache

First level cache (or *session cache*)

* Works in **session scope**
* It is **used when the objects are loaded using their primary key**
* It's **enabled** by default

*Example:*

Configuration c = new Configuration();  
c.configure("/hibernate.cfg.xml");  
SessionFactory sf = c.buildSessionFactory();  
  
Session ses1 = factory.openSession();  
Object ob1 = ses1.get(Student.class, 1);  
  
//we're in the same session, the objects will be retrieved from the cache  
Object ob2 = ses1.get(Student.class, 1);  
Object ob3 = ses1.get(Student.class, 1);  
Object ob4 = ses1.get(Student.class, 1);  
  
session.close();  
  
Session ses2 = factory.openSession();  
  
// this is a new session, the object will be retrieved from DB  
Object ob5 = ses2.get(Student.class, 1);

How To Enable Second Level Caching In Hibernate

First level cache will be enabled by default, but for enable second level cache we need to follow some settings,

* Second level cache was introduced in hibernate 3.0
* When ever we are loading any object from the database,  then hibernate verify whether that object is available in the local cache memory of that particular session [ **means first level cache** ], if not available then hibernate verify whether the object is available in global cache or factory cache [ **second level cache** ], if not available then hibernate will hit the database and loads the object from there, and then first stores in the local cache of the session [ first level ] then in the global cache [ second level cache ]
* When another session need to load the same object from the database,  then hibernate copies that object from global cache [ second level cache ] into the local cache of this new session

Second level cache in the hibernate is of  from **4** vendors…

* Easy Hibernate [EHCache] Cache from hibernate framework
* Open Symphony [OS] cache from Open Symphony
* SwarmCache
* TreeCache from JBoss

**How to enable second level cache in hibernate**

We need one provider class, here we are going to see hibernate provider class that is EHCache

**Changes required**

To enable second level cache in the hibernate, then the following **3** changes are required

* Add provider classin hibernate configuration file like…

|  |  |
| --- | --- |
|  | <property name="hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider  </property> |

Configure cache elementfor a class in hibernate mapping file…

|  |  |
| --- | --- |
|  | <cache usage="read-only" /> |

**Note**: this must write soon after <class>

* create xml file called ehcache.xml and store in at class path location [ no confusions, i mean in the place where you have mapping and configuration XML’s ] in web application.

**Important points on this second level cache**

Lets take an example, we have 2 pojo classes in our application like Student, Employee.

* If we load student object from the database, then as its the first time hibernate will hits the database and fetch this student object data and stores in the session1 cache memory [ First level cache ], then in the global cache [ second level cache ] provided if we write <cache usage=”read-only” /> in the student mapping file
* I mean hibernate will stores in the local session memory by default, but it only stores in the global cache [ second level cache ] only if we write <cache usage=”read-only” /> in the student mapping file, if not so hibernate wont stores in the global cache
* Now take another session like session 2 for example, if session 2 also load the student object then hibernate will loads from the global cache [ second level cache ] as student object is available at global [Actually when ever we want to load any object hibernate first will checks at local, then global then database right hope you remembered this ], now if session 3 modify that student object then hibernate will thorows an error because we have written <cache usage=”read-only” /> in student mapping file
* We can avoid this by writing <cache usage=”read-write” />

so remember <cache /> element has that much importance

Annotation:

actually mapping can be done using 2 ways,

* XML
* Annotations.

Actually annotations are introduced into java from JDK 1.5   
  
  
 **Now we will look how to relate  XML Mapping to Annotation**  
Mapping a class **UserDetails** to Table **USER\_DETAIL in XML --**

<class name="com.sdnext.hibernate.tutorial.dto.UserDetails" table="USER\_DETAIL">

Now Mapping a class **UserDetails** to Table **USER\_DETAIL in Annotation --**

@Entity  
@Table (name="USER\_DETAIL")  
    public class UserDetails{}

here **@Entity** declares the class as an entity (i.e. a persistent POJO class)  
**@Table** is set at the class level; it allows you to define the table, catalog, and schema names for your entity mapping. If no **@Table** is defined the default values are used: the unqualified class name of the entity.  
  
Mapping **primary key** **USER\_ID** of table to **property userId** of class **UserDetails in XML**

<id name="userId" type="long" column="USER\_ID" >

Mapping **primary key USER\_ID** of table to **property userId** of class **UserDetails in Annotation**

@Entity  
@Table (name="USER\_DETAILS")  
public class UserDetails   
{  
    @Id  
    @Column(name="USER\_ID")  
    private long userId;  
 }

here **@Id** declares the identifier property of this entity. The class **UserDetails** is mapped to the **USER\_TABLE** table, using the column **USER\_ID** as its primary key column.  
The column(s) used for a property mapping can be defined using the **@Column** annotation. Use it to override default values .  
  
**Id** Generator Class **Mapping** in **XML**

<id name="userId" type="long" column="USER\_ID" >   
      <generator class="auto"/>  
</id>

**Id** Generator Class **Mapping** in **Annotation**

    @Id  
    @Column(name="USER\_ID")  
    @GeneratedValue(strategy=GenerationType.AUTO)  
     private long  userId;  
  
Different GenerationType Enum Properties....

AUTO - either identity column, sequence or table depending on the underlying DB

TABLE - table holding the id

IDENTITY - identity column

SEQUENCE - sequence

identity copy - the identity is copied from another entity

**@GeneratedValue** Provides for the specification of generation strategies for the values of primary keys.  
**Enum GenerationType** Defines the types of primary key generation strategies.  
  
 Mapping **Column** to the **property** of class in **XML**

<property name="userName" column="USER\_NAME">

 Mapping **Column** to the **property** of class in **Annotation**

@Column(name="USER\_NAME")  
  private String userName;  
  
 @Column- provides the name of the column in a table if it is different from the attribute name. (By default, the two names are assumed to be the same.)   
  
@Transient - using when if you want skip any field of entity class to save in the database. Example-  
@Transient   
private String middleName;