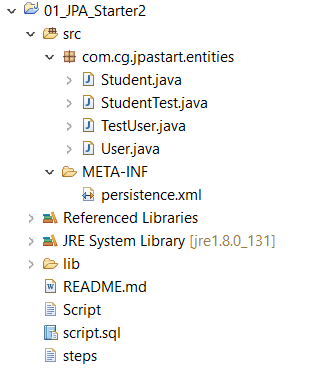
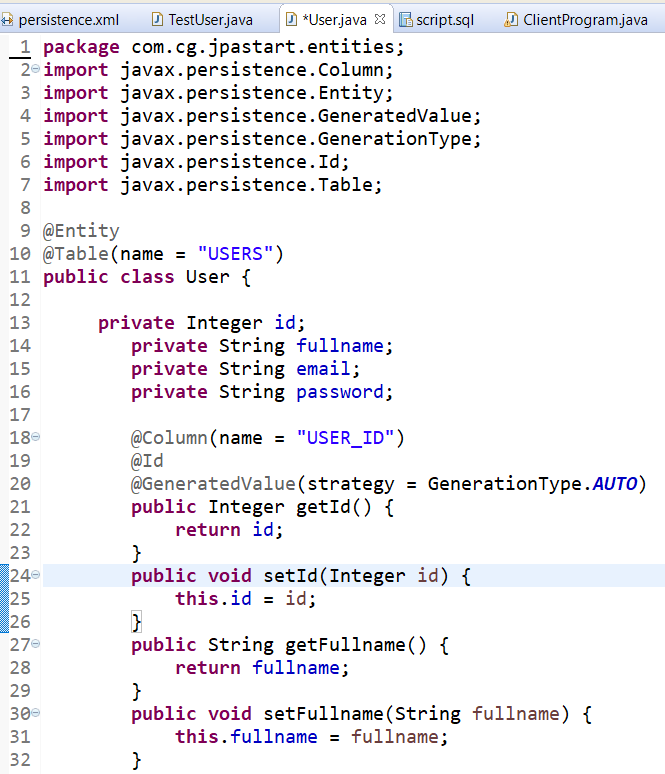


D:\EclipseNeon\Workspace\_Module4\\_MyFirstHibernatePrj



Create new java project ( not a web project )



Keep a Table Ready :

**create** **table** USERS

(

USER\_ID **int** **primary** **key** identity,

fullname **varchar**(10),

email **varchar**(20),

password **varchar**(20)

);

Identity column-autogenerated id – no need to accept data from user

Add Required Jar based on database (sqljdbc4 or ojdbc6 )

**package** com.cg.jpastart.entities;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**import** javax.persistence.Query;

**public** **class** TestUser {

**public** **static** **void** main(String[] args) {

System.***out***.println("test");

EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("JPA-PU");

EntityManager em = factory.createEntityManager();

em.getTransaction().begin();

User newUser = **new** User();

newUser.setEmail("jay@gmail.com");

newUser.setFullname("Joshi");

newUser.setPassword("jay123");

em.persist(newUser);

em.getTransaction().commit();

System.***out***.println("Added one User to database.");

Integer primaryKey = 2;

User user = em.find(User.**class**, primaryKey);

System.***out***.println("Details of user :");

System.***out***.println(user.getEmail());

System.***out***.println(user.getFullname());

System.***out***.println(user.getPassword());

String sql = "SELECT u from User u where u.email = 'jay@gmail.com'";

Query query = em.createQuery(sql);

User u1 = (User) query.getSingleResult();

System.***out***.println(u1.getEmail());

System.***out***.println(u1.getFullname());

System.***out***.println(u1.getPassword());

em.close();

factory.close();

}

}

/\* \* EntityManagerFactory is used to create an EntityManager.

An EntityManager instance is associated with a persistence context

manage entity instances and their life cycle, such as create entities,

update entities, remove entities, find and query entities.

And here are the typical steps to manage entity instances via JPA:

- Create an EntityManagerFactory from a persistence unit

- Create an EntityManager from the EntityManagerFactory

- Begin a transaction

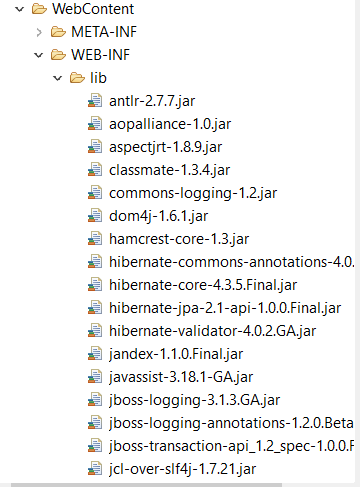
- Manage entity instances (create, update, remove, find, query, etc)

- Commit the transaction

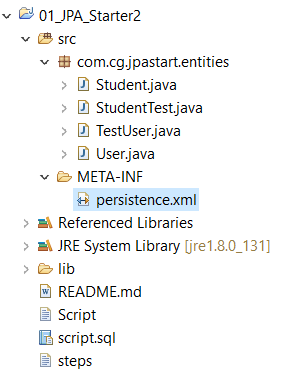
- Close the EntityManager and EntityManagerFactory

\*/

Add hibernate related libraries ( from smita / cb sir project



Add Driver jar ( based on database -ojdbc6 or sqljdbc4.jar)



<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<persistence xmlns=*"http://java.sun.com/xml/ns/persistence"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/persistence*

*http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"*

version=*"2.0"*>

<persistence-unit name=*"JPA-PU"* transaction-type=*"RESOURCE\_LOCAL"*>

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<!-- <properties>

<property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect"/>i added bec of 12c connection error

<property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:scott@//localhost:1521/pdborcl" />

<property name="javax.persistence.jdbc.user" value="scott" />

<property name="javax.persistence.jdbc.password" value="tiger" />

<property name="javax.persistence.jdbc.driver" value="oracle.jdbc.driver.OracleDriver" />

</properties> -->

<properties>

<property name=*"hibernate.dialect"* value=*"org.hibernate.dialect.SQLServerDialect"*/>

<property name=*"javax.persistence.jdbc.url"* value=*"jdbc:sqlserver://localhost:1433;databaseName=TSQL2012"* />

<property name=*"javax.persistence.jdbc.user"* value=*"sa"* />

<property name=*"javax.persistence.jdbc.password"* value=*"Ritwik123"* />

<property name=*"javax.persistence.jdbc.driver"* value=*"com.microsoft.sqlserver.jdbc.SQLServerDriver"* />

</properties>

</persistence-unit>

</persistence>

Difference between Hibernate and JPA

JPA is specification and hibernate is implementation

**JPA** or Java Persistence API is a **standard specification** for ORM implementations whereas **Hibernate** is the actual **ORM implementation** or framework.

As the definition says its API, it is only the specification. There is no implementation for the API. JPA specifies the set of rules and guidelines for developing the interfaces that follows standard. **Straight to the point : JPA is just guidelines to implement the Object Relational Mapping (ORM)  and there is no underlying code for the implementation**.

Hibernate is the most popular **JPA provider**.

|  |  |
| --- | --- |
| **ORM Softwares** | **Vendor Names** |
| Hibernate | JBOSS |
| IBATIS | Apache |
| TopLink | Oracle Corporation |
| OJB(Object Java Bean) | Apache |
| JDO(Java Data Object) | Adobe |

**Hibernate can be used in JPA**

**If you are using Hibernate's proprietary API, you'll need the hibernate.cfg.xml.**

**SessionFactory and Session are hibernate-specific.**

**JPA is standard ->**

**If you are using JPA i.e. Hibernate EntityManager, you'll need the persistence.xml.**

**Prefer EntityManagerFactory and EntityManager. They are defined by the JPA standard.**

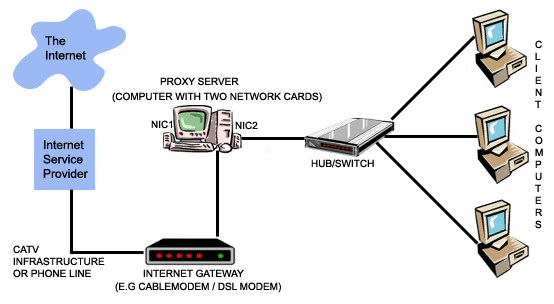
**Using EntityManagerFactory approach allows us to use callback method annotations like @PrePersist, @PostPersist,@PreUpdate with no extra configuration.**

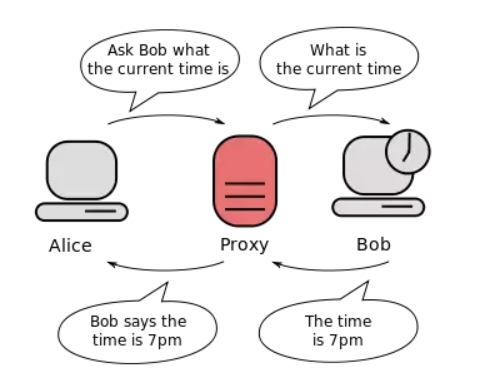
**Using similar callbacks while using SessionFactory will require extra efforts.**

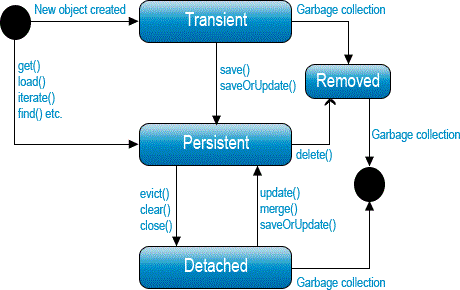
Entity manager is a part of JPA specification while Hibernate implements its own solutions based on Session object, to handle persistence. JPA's elements are normally the **interfaces, hibernate uses classes**

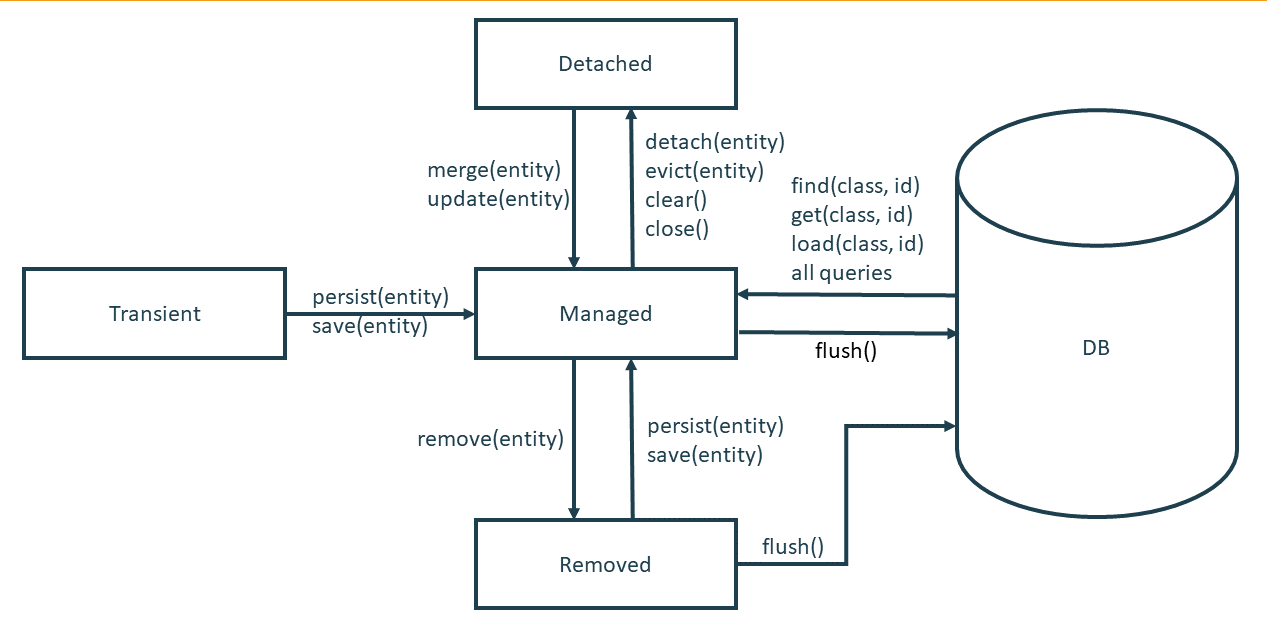
**Object Life Cycle:**

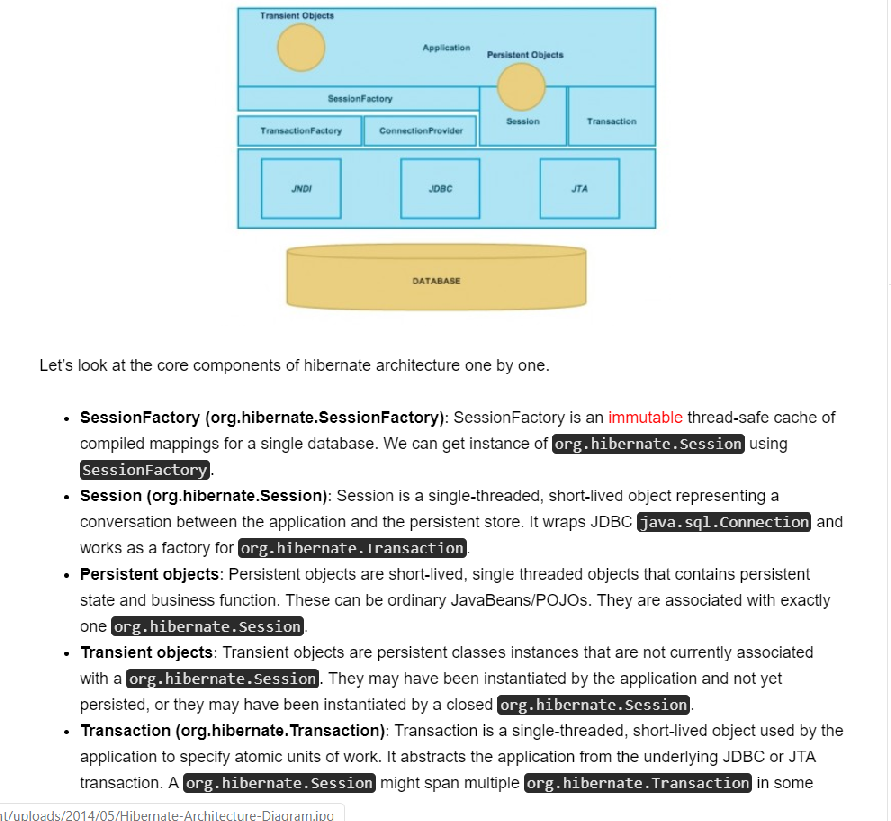
A proxy server is a dedicated computer or a software system running on a computer that acts as an intermediary between an endpoint device, such as a computer, and another server from which a user or client is requesting a service.

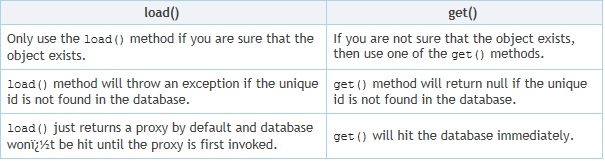


****



****

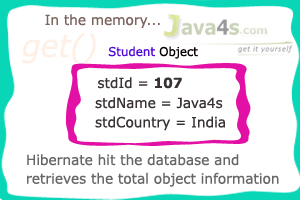
****

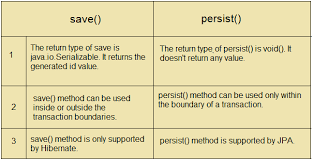


**The entity can be retrieved in 2 ways.**

load() - returns the proxy object with an identifier.

get() - returns the complete object from database.



****

## Caching

Caching is a facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate achieves the second goal by implementing first level cache.

Caching structurally implies a **temporary store** to keep data for **quicker access later on**.

**Second level shared cache** is an auxiliary technique, mainly used in JPA, **to enhance performance;** it is used especially **during large inflow, outflow of data between the database and the application.** Caching also **reduces search time when the searched entity is already in the cache, otherwise it is fetched from the database to serve the purpose.** However, when a subsequent search query is fired it takes little time as the searched entity is already in the cache

**Fist level cache** in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.

Entity Manager always maintains a cache called the first level cache during all crud operation.

**it is associated with Session object**. As we know session object is created on demand from session factory and**it is lost, once the session is closed**. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is **not accessible to any other session object** in any other part of application.

1. First level cache is associated with “session” object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. First level cache is enabled by default and you can not disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
6. The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
7. The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

**Example**

In first level cache CRUD operations are performed per transaction basis to reduce the number of queries fired to the database.

That is, an entity modified several times within the same transaction is done in the cache only, modification at the database level is slated until final UPDATE statement is fired at the end of the transaction.

JPA entities are cached at the persistence context level and guarantees that there will be one object instance per persistence context for a specific row of a database table. Concurrent transactions affecting the same row are managed by applying an appropriate locking mechanism in JPA.

1. **Second Level Cache**: Hibernate Second Level cache is disabled by default but we can enable it through configuration. Currently EHCache and Infinispan provides implementation for Hibernate Second level cache and we can use them

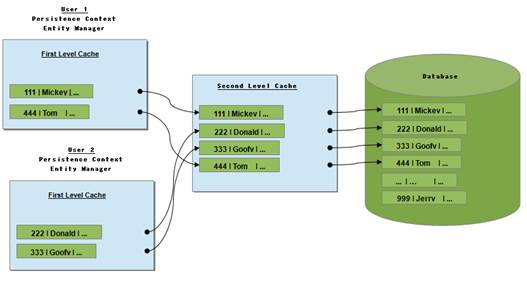
## First level Cache

## Second Level Cache

This level of cache emerged more due to performance reasons than absolute necessity; in fact adding more shared cache increases the possibility of the problem of a stale read. As illustrated in **figure 3**, second level cache sits between Entity Manager and the database. Persistence context shares the cache, making the second level cache available throughout the application. Database traffic is reduced considerably because entities are loaded in to the shared cache and made available from there. So in a nutshell second level cache provides the following benefits.

* Persistence provider manages local store of entity data
* Leverages performance by avoiding expensive database calls
* Data are kept transparent to the application
* CRUD operation can be performed through normal entity manager functions
* Application can remain oblivious of the underlying cache and do its job inadvertently

With Java EE 7, JPA 2.0 acknowledged the need of a second level cache and has provided very minimal APIs. Perhaps, it is still undecided whether JPA would fully support a functional second level cache in the near future or not. However, persistence providers like EclipseLink, Hibernate, etc. have their own set of implementation created either reusing an existing one or developing their own from scratch. As a result, a large part of the shared cache API is not standardized. The good news is JPA 2.0 nodded at last and we shall wait to see its future course in its specification.



## Implementing Second Level Cache

The javax.persistence.Cache interface of the persistence provider can be used to interact with the second level cache. This interface provides functions such as, contains: to check whether the cache contains a given (as parameter) entity, variation of evict: to remove a particular entity from the cache, evictAll: to clear the cache and unwrap: to return specified cache implementation by the provider (ref. Javadoc JPA API). We can use @Cacheable annotation to make a POJO eligible to be cached. In this way the persistence provider knows which entity is to be cached. If no such annotation is supplied then entity and its state is not cached by the provider. @Cacheable takes a parameter of boolean value, default is true.

1. **Query Cache**: Hibernate can also cache result set of a query. Hibernate Query Cache doesn’t cache the state of the actual entities in the cache; it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.

### Hibernate Caching – First Level Cache Example

**Listing 1: using @Cacheable annotation**

**package** org.mano.dto;

@Entity

@Cacheable(**true**)

**public** **class** Account{

        @Id @GeneratedValue(strategy=GenerationType.AUTO)

**private** Long accountNumber;

**private** String name;

        @Temporal(TemporalType.DATE)

**private** Date createDate;

**private** Float balance;

        //... constructors, getters, setters

}

Once the entity is all set to be cache-able, we need to provide an appropriate caching mechanism to be used by the persistence provider in persistence.xml. This is done by setting one of the following values in shared-cached-mode element such as:

<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.1" xmlns="...">

<persistence-unit name="JPALockingDemo" transaction-type="RESOURCE\_LOCAL">

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

<class>org.mano.dto.Account</class>

**<shared-cache-mode>ALL</shared-cache-mode>**

<properties>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bank"/>

<property name="javax.persistence.jdbc.user" value="bank"/>

<property name="javax.persistence.jdbc.password" value="secret"/>

<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>

<property name="javax.persistence.schema-generation.database.action" value="create"/>

</properties>

</persistence-unit>

</persistence>

|  |  |
| --- | --- |
| **Value** | **Description** |
| **ALL** | All entities are cached |
| **NONE** | Disable caching |
| **ENABLE\_SELECTIVE** | Enable caching only for those entities which has @Cacheable(true) |
| **DISABLE\_SELECTIVE** | Enable caching only for those entities which are not specified with @Cacheable(false) |
| **UNSPECIFIED** | Applies persistence provider-specific default behavior |

The POJO Account in **Listing 1** is now cache-able, code in **Listing 2** will demonstrate the case.

**Listing 2: Demonstrating the use of cache-able POJO**

**package** org.mano.app;

**public** **class** Main {

**public** **static** **void** **main**(String[] args) {

                EntityManagerFactory factory = Persistence

                               .createEntityManagerFactory("JPALockingDemo");

                EntityManager manager = factory.createEntityManager();

                Account account = **new** Account(111l,"Patrick",**new** Date(),23000f);

                manager.getTransaction().begin();

                manager.persist(account);

                manager.getTransaction().commit();

                Cache cache = factory.getCache();

                System.out.println("cache.contain should return true: "+cache.contains(Account.**class**, account.getAccountNumber()));

                cache.evict(Account.**class**);

                System.out.println("cache.contain should return false: "+cache.contains(Account.**class**, account.getAccountNumber()));

                manager.close();

                factory.close();

        }

}

## Conclusion

JPA caching is flexible enough to configure per class basis or globally with the help of persistence unit settings or class settings. We can either set shared-cache-mode element in the persistence.xml or dynamically set javax.persistence.sharedCache.mode property while creating entity manager factory. In concurrent transactions, setting the cache mode to NONE in view of better performance may on the contrary lead to performance slowdown and is not recommended; such situations should rather be handled with an appropriate locking mechanism.