Hibernate Criteria , NamedQuery, NamedQueries, NativeQuery, NamedNativeQuery

Select \* from question;

Select q from question q; (JPQL & HQL)

createQuery()

**Hibernate NativeSQL**

Hibernate provides the option to execute native SQL queries through the use of the **SQLQuery** object. The Session.createNativeQuery(String query) method is used to create the [NativeQuery](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/query/NativeQuery.html" \t "_blank) object and execute it.

**Example:**

Query query = session.createNativeQuery("SELECT id, name, age FROM student");

// Get All Students

List<Object[]> rows = query.getResultList();

for(Object[] row : rows){

Student stud = new Student();

stud.setId(Long.parseLong(row[0].toString()));

stud.setName(row[1].toString());

stud.setAge(Long.parseLong(row[2].toString()));

System.out.println(stud);

}

The getResultList() method returns a List of Object arrays, which we need to explicitly parse to double, long etc. Hibernate will use **ResultSetMetadata** to deduce the actual order and types of the returned scalar values.

To avoid the overhead of using ResultSetMetadata, or simply to be more explicit in what is returned, one can use addScalar():

session.createNativeQuery("SELECT \* FROM student")

.addScalar("id", LongType.INSTANCE)

.addScalar("name", StringType.INSTANCE)

.addScalar("age", LongType.INSTANCE);

The addEntity() method used to get entity objects from a Native SQL query.

Query<Student> studentSQLQuery = session.createNativeQuery("SELECT id, name, age FROM student").addEntity(Student.class);

Alternatively, this can be provided as an argument to the overloaded method:

Query<Student> studentSQLQuery = session.createNativeQuery("SELECT id, name, age FROM student", Student.class);

The addJoin() method used to fetch the data from associated table using tables join.

Query<Student> query = session.createNativeQuery("SELECT s.id, s.name, a.\* FROM student s JOIN address a ON s.id = a.stud\_id")

.addEntity("e", Student.class)

.addJoin("a", "student.address");

<https://www.objectdb.com/java/jpa/query/named>

## Named Queries

In Hibernate, a [Named Query](https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/NamedQuery.html) is a SQL expression with a predefined unchangeable query string. We can define a named query either in the **hibernate mapping file** or in an **entity class**.

**Annotations used in an entity class:**

* @NamedQueries - used to define the multiple HQL expressions.
* @NamedQuery - used to define the single HQL expression.
* @NamedNativeQueries - used to define the multiple native SQL expressions.
* @NamedNativeQuery - used to define the single native SQL expression.

@NamedQuery has two attributes:

* **name** - used to specify a name by which a session object can locate the query.
* **query** - used to specify the HQL statments.

**Example:**

@Entity

@Table(name = "student")

@NamedQueries({

@NamedQuery(name = "viewAllStudents", query = "FROM Student s")

@NamedQuery(name = "findStudentByID", query = "FROM Student s WHERE s.id = :id")

@NamedQuery(name = "findStudentByName", query = "FROM Student s WHERE s.name = :name")

})

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "email")

private String email;

public student(){

}

public Student(String firstName, String lastName, String email) {

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

//getters and setters

}

The session.getNamedQuery(String query\_name) method used to run the specified named query defined at the entity class.

public class Application {

public static void main(String[] args)

{

StandardServiceRegistry registry = new StandardServiceRegistryBuilder().configure().build();

SessionFactory sessionFactory = new MetadataSources( registry ).buildMetadata().buildSessionFactory();

Session session = sessionFactory.openSession();

session.beginTransaction();

Query query = session.getNamedQuery("findStudentByID").setParameter("id", 32);

List students = query.getResultList();

for(Student student : students)

{

System.out.println(student);

}

session.getTransaction().commit();

session.close();

}

}

## Caching in Hibernate

Hibernate performs caching to **optimize the performance** of an application. It's used to reduce the number of database hits by storing data locally in a cache.

### First-Level Cache / L1 cache

The first-level cache is the first place that Hibernate checks for cached data. L1 caching in hibernate is enabled by default, and we can't disable it. It is a mandatory cache through which all requests must pass.

* This type of cache is associated with the **Session object**. Each Session object caches data independently, so there is no sharing of cached data across sessions, and the cached data is deleted when the session closes.
* An L1 cache is internal to a Session object and not accessible to any other session object in an application.
* This type of cache is only useful for repeated queries in the same session.
* When we query an entity first time, it retrieves the data from the database and stores it in the L1 cache associated with the session. If we query again with the same session object, then it loads the data from the L1 cache.

### Second-Level Cache/ L2 Cache

L2 cache is responsible for caching objects and sharing data across sessions. The L2 cache is associated with the **SessionFactory object** and is shared among all sessions created using the same session factory.

If the requested query results are not in the first-level cache, then the second-level cache is checked. Any technology that supports out-of-the-box integration with Hibernate can be plugged in to act as a second-level cache.

The L2 cache is disabled by default but we can enable it through configuration. If L2 caching is enabled for an entity, the following workflow is used:

* If an instance is already present in the L1 cache, then it is returned from there.
* If an instance is not found in the L1 cache, the L2 cache is searched, and if found the data is fetched and returned from there.
* If the data is not cached in the L2 cache, then the necessary data is loaded from the database and an instance is assembled and returned.

We can enable the L2 cache by adding the following properties to the **hibernate configuration file**.

<!-- Enable the second-level cache -->

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

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

To make an **entity** eligible for L2 caching, we annotate it @Cache annotation and specify a **cache concurrency strategy**. It's a best practice to annotate the entity class with @Cacheable annotation, although not required by Hibernate. So the entity class looks like:

@Cacheable

@Cache(usage = CacheConcurrencyStrategy.READ\_ONLY)

@Entity

@Table(name="student")

public class Student

{

//code goes here...

}

A concurrency strategy it instructs hibernate on how to store objects in the cache and retrieve them, in an environment where multiple sessions might be trying to manipulate the same object simultaneously. If we enable an L2 cache, then we decide the cache concurrency strategy for each persistent class and collection. The cache concurrency strategies are:

* **READ\_ONLY** - Use this strategy only for entities where we never change any data and use data as a reference.
* **NONSTRICT\_READ\_WRITE** - Doesn't guarantee the consistency between the cache and the database. Use this strategy only for entities where we change data rarely.
* **READ\_WRITE** - Use this strategy only for entities where we read and update data.
* **TRANSACTIONAL** - Use this strategy to cache the full transactions made on the entity.

<https://www.javatpoint.com/hibernate-second-level-cache>

**Object States in Hibernate**

The three Object States in Hibernate are:

* **Transient State**
* **Persistent State**
* **Detached State**

An object of a persistent class (a class mapped to a relational database table) can be in one of three different states. These states are defined in relation to a **persistence context** (Session object).

**Transisent**

* When an object is created using the new operator and not yet associated with a Hibernate Session, then the object state is transient.
* It doesn't represent a row in the database.
* Transient instances are garbage collected if the application does not hold a reference anymore.

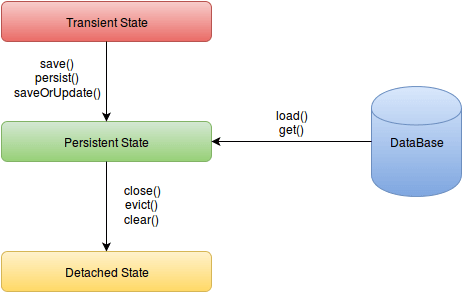
**Persistent**

* The object state is persistent when it is associated with the hibernate session.
* The Persistent object represents a row in the database and has an identifier value.
* Transient instances can be made persistent by associating them with a Session.
* The save(), persist() and saveOrUpdate() methods are used to associate a transient object with a session and make them persistent.
* Hibernate detects the changes made to persistent objects and synchronizes the state with the database.
* Whenever we get the data from the database using get() or load() methods, the data will be in the persistent state.

**Detached**

* When a persistent object has its session closed, then it becomes detached.
* Any changes made to detached objects will not be saved automatically to the database.
* When a detached instance reattached with a new Session at a later point in time, it makes the object persistent again.
* The Session class' close(), evict(Object), and clear() methods are used to move a persistent object to the detached state.
* The Session class' update(Object) and merge(Object) methods can used to reattach detached objects to a session.

**Object states:**



## HQL

**H**ibernate **Q**uery **L**anguage is the object-oriented query language of the Hibernate Framework. HQL is very similar to SQL except that we query against **persistent objects** instead of tables and columns.

Hibernate then **translates** the HQL queries to SQL queries and performs the action on the database. In hibernate, we can directly write SQL statements using Native SQL, but using HQL or JPQL is recommended. From the developer's perspective, this separates the application layer from the persistence layer and abstracts away the actual database interaction, promoting flexibility and reusability.

HQL is **case-sensitive for properties** like table and column names and not for keywords like SELECT, FROM, and was, etc.

**Advantages of HQL:**

* It supports OOP concepts like polymorphism, inheritance, and abstraction.
* It is database-independent and easy to learn.

### HQL Examples

* HQL **Select** Query example to retrieve a student details whose id is 101.

TypedQuery<Student> query = session.createQuery("FROM Student WHERE id = '101' ", Student.class);

List<Student> students = query.getResultList();

* HQL **Update** Query example to update the name to "John" whose id is 105.

Query query = session.createQuery("UPDATE Student SET name = :stud\_name WHERE id = :stud\_id");

query.setParameter("stud\_name", "John");

query.setParameter("stud\_id", "105");

int result = query.executeUpdate();

Please note that HQL should only be used to batch-update records. If you are updating a single record, it's preferable to update the actual Java object's properties, and then persist that object and its changes back to the database with session.update(object) or session.flush()

* HQL **Delete** Query example to delete a student whose id is 108.

Query query = session.createQuery("DELETE Student WHERE id = :stud\_id");

query.setParameter("stud\_id", "108");

int result = query.executeUpdate();

Similarly to updates, HQL should only be used to batch-delete records. If you are deleting a single record, it's preferable to use session.delete(object).

* HQL **Insert** statement cannot insert values directly in to a table. It is only used to insert rows from another table. It supports only INSERT INTO … SELECT … .

Query query = session.createQuery("INSERT INTO Student(id, name) SELECT s\_id, s\_name FROM NewStudent");

int result = query.executeUpdate();

# Object Relational Mapping (ORM)

Most object-oriented applications use a relational database to store and manage the application data. The relational database represents data in a table, whereas the data in object-oriented applications encapsulated in an object.

We can access a class by using its objects. However, to access the tabular data, we need to use a query language. Using tabular data in an object-oriented application requires a conversion between the two types of data.

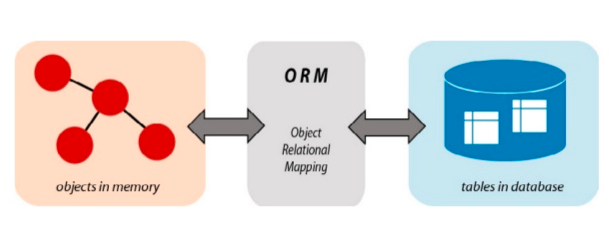
As a result, it is not possible to store the objects directly in a relational database. These differences between object-oriented and relational database paradigms are called **impedance mismatch**.

[Impedance Mismatch](https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch) can exist at the following points:

* **Granularity** - refers to the mismatch in the number of classes that are mapped with a certain number of tables in the database.
* **Inheritance** - Java classes in the application are commonly related to each other through an inheritance hierarchy. However, the tables within the database can't be represented through an inheritance hierarchy.
* **Identity** - The relational database distinguishes an object instance on the basis of their primary key. However, an object model distinguishes an object on the basis of object identity and object equality.
* **Association** - In the object model, two classes are linked by association. However, in relational databases, the linking of tables is achieved with the help of foreign keys.
* **Navigation** - The ways of accessing objects in Java and in RDBMS are fundamentally different.

To solve the impedance mismatch, we use an **ORM** tool that converts the **data between relational databases and object oriented programming languages**.

[ORM](https://en.wikipedia.org/wiki/Object-relational_mapping) stands for **O**bject-**R**elational **M**apping, uses objects to connect the Object-Oriented programming language and the database systems, which facilitates the SQL to work along with the object-oriented programming concepts.



### Benefits of ORM

* ORM maps an object to the table.
* We can hide the details of SQL queries from OO logic. This propagates the idea of data abstraction.
* It provides methods for automatic versioning and timestamping.
* It provides caching support for better performance.
* Best suited for large projects
* Injected transaction management
* Configurable logging
* Faster development of applications

There are lots of ORM tools available such as Hibernate, JPA, Active JPA, iBATIS, IBM Pure Query, etc.

Project1 [Javalin project – Postgres running in AWS RDS]

(Maven Project)

Src/main/java

Com.revature.main – Starter.java

Com.revature.model – All the Entity Bean class

Com.revature.controller -- All the controllers class

Com.revature.dao -- All the Data Access Object Class

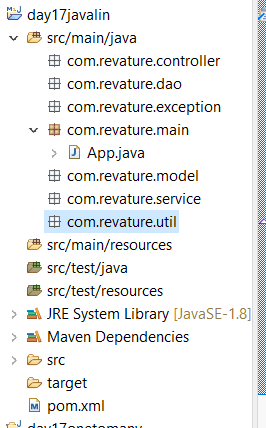
Com.revature.service -- All the Service Classes

Com.revature.exception – All type exception class

Com.revature.util --- HibernateUtil, ConnectionUtil

Src/main/resources - properties file, any other static files, (public,template,static folders)

Src/test/java --- All Junit test case code



Technologies Used

* 1. Javalin – For creating the Rest End points
  2. Junit, Selenium & Cucumber for Testing
  3. Hibernate – ORM Framework (DB Interaction)
  4. AWS RDS – Postgres db (or) maria db
  5. Logging