TEKsystems | PER SCHOLAS

A Partnership for Progress

Object-Relational Mapping
Java Persistence API
Hibernate

ORM - Object-Relational Mapping, JPA - Java Persistence API, Hibernate

Learning Objectives:

In this presentation, we will get to know the Java persistence standard based on Hibernate.

By the end of this session, learners will be able to:

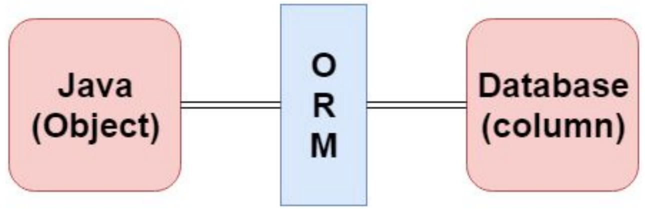
- Explain Object-Relational Mapping (ORM).
- Explain the Hibernate Application Architecture and Hibernate - Annotations.
- Describe how to use JPA to store and manage Java objects in a relational database.
- Demonstrate the JPA with Hibernate as Implementation.
- Demonstrate setting up a Hibernate project.
- Describe Hibernate Query Object (HQO) and Hibernate Query Language (HQL).
- Demonstrate how to use Hibernate Query Object (HQO).

Outline

Object Relational Mapping (ORM)

Object Relational Mapping (ORM) is the concept/process of converting data from object-oriented language to a relational database, and vice versa.

ORM translates the programming code attributes (variables) into the columns in the table. It is good for managing various database operations, such as insertion, update, and deletion effectively.



The mapping between the object model and the relational database should be as follows:

- Class <-> Table.
- Java Object <-> Row.
- Class Attribute <-> Column.

Object Relational Mapping (ORM) (continued)

What is JPA?

- Java Persistence API (JPA) is the Object-Relational Mapping (ORM) standard for storing, accessing, and managing Java objects in a relational database.
- JPA is only a specification; it is not an implementation, but several implementations are available. Popular implementations include **Hibernate**, **EclipseLink**, **Apache OpenJPA**, and many more.
- JPA permits the developer to work directly with objects rather than with SQL statements. The JPA implementation is typically called a *persistence provider*.
- JPA is the EE standard specification for ORM in Java EE.
- JPA API is a set of rules and a framework to set interfaces for implementing ORM.

Object Relational Mapping (ORM) (continued)

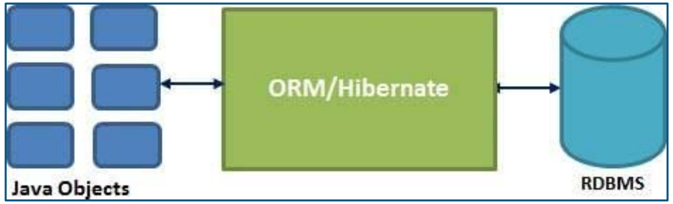
- **Problem:**
  - ➢ Database interaction requires a great deal of code, and is therefore a burden on data maintenance. Database interaction also requires a proprietary framework.

- **Solution:**
  - ➢ Most contemporary applications use relational database to store data. Recently, many vendors switched to relational database to reduce their burden on data maintenance. This means that object database or object relational technologies are taking care of storing, retrieving, updating, and maintenance. The core part of the object relational technologies is the mapping file. As mapping file does not require compilation, we can easily make changes to multiple data sources with less administration.

  - ➢ ORM is a programming ability or approach to mapping **Java objects** to **database tables,** and vice versa. JPA is one possible approach to ORM. By using JPA, the developer can **map, store, update, delete, and retrieve** data from relational databases to Java objects, and vice versa.

What is Hibernate?

- **Hibernate** is an implementation of JPA API and uses ORM techniques. In that, we can use the standard JPA API, and configure applications to use Hibernate as the provider of the specification under the covers. Hibernate provides more features beyond what JPA specifies.
- Hibernate maps **Java classes to database tables**, and from **Java data types to SQL data types.** Hibernate maps also relieve the developer from the majority of common data persistence-related programming tasks. This is especially beneficial for developers with limited knowledge of SQL.
- The Hibernate framework consists of several components such as ***Hibernate ORM, Hibernate Search, Hibernate Validator, Hibernate CGM, and Hibernate Tools***. In this course, we will use Hibernate ORM, which is the core component of the Hibernate framework for mapping Java model classes.
- JDBC is not object-oriented; rather, we are dealing with values means of primitive data. In Hibernate, each record is represented as a **Object,** but in JDBC, each record is no more than data, which is nothing but primitive values.



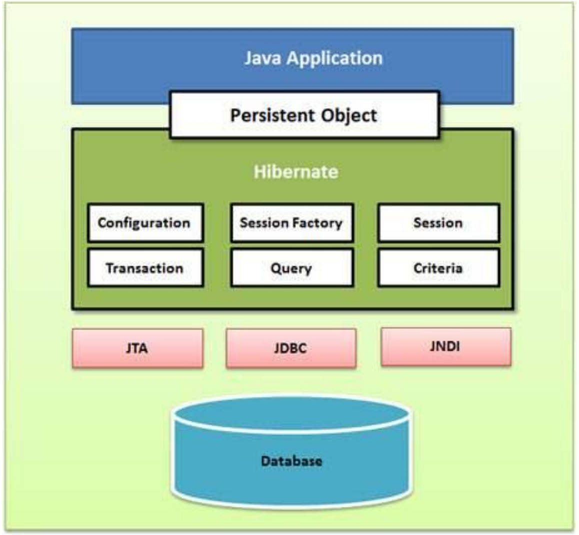
Hibernate With JPA

- JPA is a specification for persistence providers to implement. Hibernate is one such implementation of JPA specification. We can annotate our classes as much as we like with JPA annotations; however, without an implementation, nothing will happen.
- Think of JPA as the guidelines/specification that must be followed, or as an interface while Hibernate's JPA implementation is code that meets the API, as defined by JPA, providing the "under-the-hood" functionality.
- When we use hibernate with JPA, we are actually using the Hibernate JPA implementation. The benefit of this is that we can swap out Hibernate's implementation of JPA for another implementation of the JPA specification.
- When we use straight Hibernate, we lock into the implementation because other ORMs may use different methods/configurations and annotations; therefore, we cannot just switch over to another ORM.

Hibernate Application Architecture

- Hibernate has a layered architecture, which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (persistent objects) to the application.
- Let's understand what each block represents in detail:

  Persistent Objects: Plain Old Java Objects (POJOs), which get persisted as one of the rows in the related table in the database by Hibernate. They can be configured in configuration files **(hibernate.cfg.xml or hibernate.properties)** or annotated with **@Entity annotations**.



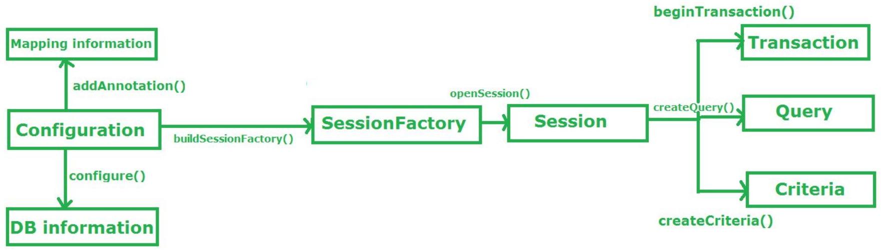


Fig: Hibernate Architecture

Persistent Objects

Persistent Objects:

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate. Hibernate works best if these classes follow some simple rules, also known as the **POJO Programming Model**.

These are the some rules of persistent classes; however, none are hard requirements:

- All Java classes that will be persisted need a default constructor.
- All classes should contain an ID in order to allow easy identification of objects within Hibernate and the database. This property maps to the primary key column of a database table.
- All attributes that will be persisted should be declared private and have **getXXX** and **setXXX** methods defined in the JavaBean style.
- A central feature of Hibernate, called *proxies*, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.

Configuration

Configuration:

Configuration is a class, which is present in the **org.hibernate.cfg** package. The configuration object is the first Hibernate object; it activates the Hibernate framework, and reads both **configuration files and mapping files**. It is usually created only once during application initialization, and represents a configuration file (`hibernate.cfg.xml`) or properties file required by Hibernate.

The configuration object provides two keys components:

- Database Connection − Handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** or **hibernate.cfg.xml.**
- Class Mapping Setup − This component creates the connection between the Java classes and database tables.

```
Configuration cfg = new Configuration();  //This line activate Hibernate Framework

cfg.configure(); // read both cfg file and mapping files
```

Session Factory in Hibernate

SessionFactory() Object:

- SessionFactory is an interface, which is present in the **org.hibernate** package.
- **Configuration object is used to create a SessionFactory object.** The SessionFactory is a thread-safe object and is used by all of the threads of an application.
- **It is usually created during application start-up and kept for later use.** We would need one SessionFactory object per database, using a separate configuration file. So, if you are using multiple databases, you would have to create multiple SessionFactory objects.
- The SessionFactory in Hibernate is responsible for the creation of Session objects.

```
        SessionFactory factory=cfg.buildSessionFactory();
//      buildSessionFactory() method gathers the meta-data which is in the cfg Object.
        From cfg object it takes the JDBC information and create a JDBC Connection.
```

Session in Hibernate

Session() Object:

- **Session** is an interface, which is present in the **org.hibernate package.** A **Session** is used to get a physical connection with a database. A Session object is created based upon a ***SessionFactory()*** object.
- A **Session()** is designed to be instantiated each time an interaction is needed with the database.
- The **Session** objects should not be kept open for a long time because they are not usually thread-safe, and they should be created and destroyed as needed**.**
- **A Session object is used to perform Create, Read, Update, and Delete(CRUD) operations for instances of mapped entity classes.** Instances may exist in one of three states:
  - **Transient**: never persistent, not associated with any Session.
  - **Persistent**: associated with a unique Session.
  - **Detached**: previously persistent, not associated with any Session.

```
Session session = factory.buildSession();
```

Session in Hibernate (continued)

Important "Session Methods"

Session.save() method:

The **save()** method saves the transient entity. Before saving, it generates an new identifier and INSERT record into a database when the transaction is committed. This method is used to bring only a transient object to a persistent state. There are two overloading functions of this method:

- **save(Object object)**: Generates a new identifier and INSERT record into a database. The insertion fails if the primary key already exists in the table.
- **save(String entityName, Object object)**: Accepts the entity name and instance of entity.

Session.update() method:

The **update()** method updates the entity for persistence, using the identifier of the detached object or the new instance of entity created with existing identifier. If the object is already in the session with the same identifier, it throws an exception. This method updates the associated object if the cascade is defined as "save-update."

- **update(Object object)**: Accepts the instance of the new entity or any detached object from the session.
- **update(String entityName, Object object)**: Accepts the entity name and instance of object.

Session in Hibernate (continued)

Important "Session Methods"

Session.merge() method:

The **merge()** method is used to merge an object with a persistent object on the basis of the same identifier. The object, as an argument is not changed, and the method returns persistent object:

1. The state of the object passed as an argument is copied to the object in the Hibernate session with the same identifier, and returns the persistent object.

2. If the object is not already present in the session with the same identifier as passed in the argument, the session will first load the object for the identifier of the object passed as an argument, and then merge it and return the persistent object of the session.

3. If the persistent object for the identifier of the object passed as an argument is not already in session and database, then a copy of object passed as an argument is persisted and is returned.

merge(Object object): Accepts the entity object and returns persistent object.
merge(String entityName, Object object): Pass entity name and entity object.

Visit the link below for more information:

https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Session.html

Session in Hibernate (continued)

Important "Session Methods"

Session.delete() method or Session.remove() method

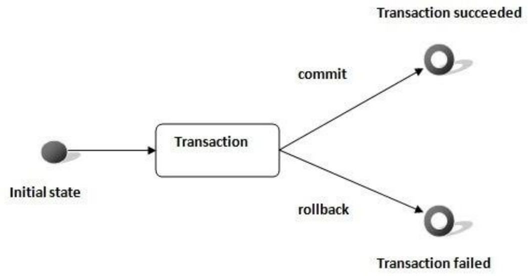
In Hibernate, an entity can be removed from a database by calling the `Session.delete()` or `Session.remove()`. Using these methods, we can remove a transient or persistent object from datastore.

Transaction

Transaction: Enables you to achieve data consistency and rollback in case something unexpected happens.

A transaction can be described by Atomicity, Consistency, Isolation and Durability (ACID) properties. It maintains abstraction from the transaction implementation (JTA,JDBC).

```
Transaction tx=session.beginTransaction();
tx.commit();
```



The methods of Transaction interface are as follows:

1. void begin() starts a new transaction.
2. void commit() ends the unit of work unless we are in ***FlushMode.NEVER***.
3. void rollback() forces this transaction to rollback.
4. void setTimeout(int seconds) sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. boolean isAlive() checks if the transaction is still alive.
6. void registerSynchronization(Synchronization s) registers a user synchronization callback for this transaction.
7. boolean wasCommited() checks if the transaction is committed successfully.
8. boolean wasRolledBack() checks if the transaction is rolled back successfully.

Query Object

Query Interface:

- Query is an interface that presents inside **org.hibernate package.**
- A Query **instance** is obtained by calling ***Session.createQuery()*** method.
- Query objects use **SQL** or **Hibernate Query Language (HQL)** to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally, to execute the query.

```
Query query=session.createQuery();
```

Entity/POJO/Model Classes

- Entity is a Java object that is going to be persisted. Entity classes are decorated with Java annotations such as @Entity, @Id, @Table, or @Column.
- Entities are nothing but beans or Models, and they contain default **constructor**, **setter**, and **getter** methods for the attributes or private variables (class variables).

Hibernate Application Stages

- Phase 1 Composed of...
  - POJO Classes:
    - Creating Entities - Entities are nothing but beans or Models, and they contain default constructor, setter, and getter methods of those attributes.
      (Note: We will be handling this via annotations.)
  - DAO/Service Classes containing service methods:
    - Create, Update, Delete, or Find a record.
- Phase 2 Composed of…
  - Mapping File: Hibernate Configuration:
    - Configurations - Database connection information, schema level settings, and entity mapping, using the mapping file (**hibernate.properties** or **hibernate.cfg.xml**).

Setting up a Hibernate

1. Create Java Maven Project.
2. Configure Maven Dependencies for Hibernate and Database in pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>5.5.7.Final</version>
</dependency>

<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-annotations</artifactId>
   <version>3.5.5-Final</version>
</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.25</version>
</dependency>
```

Maven automatically downloads the required JAR files.

Hibernate - Annotations

Annotations:

- We can write configurations using **annotations**. The annotations are used for classes, properties, and methods. Annotations start with the **'@'** symbol, and are declared before the class, property, or method is declared.
- Hibernate Annotations is a powerful way to provide the metadata for the Object and Relational Table mapping.

Annotations used in our examples:

Annotation	**Description**
@Entity	This annotation indicates that the class is mapped to a database table. By default, the ORM framework understands that the class name is the same as the table name. The @Entity annotation must be placed before the class definition.
@Table	This annotation is used if the class name is different than the database table name, and it must be placed before the class definition.
@Id	This annotation specifies that a field is mapped to a primary key column in the table.
@Basic	This annotation tells JPA the below variable is a regular attribute.

Hibernate - Annotations (continued)

Annotation	Description

@Column	This annotation is used to map an instance field of the class to a column in the database table, and it is placed before the getter method of the field. By default, Hibernate can implicitly infer the mapping based on field name and field type of the class, but if the field name and the corresponding column name are different, we have to use this annotation explicitly:
	
	`@Column(name = "ACC_NO", unique = false, nullable = false, length = 100)`
	`private String accountNumber;`

@JoinColumn	This annotation is used to map the foreign key column of a managed association (e.g., one-to-one, many-to-one, many-to-many).
@TableGenerator	This annotation is used to specify the value generator for the property specified in @GeneratedValue annotation. It creates a table for value generation.
@ColumnResult	This annotation references the name of a column in the SQL query using the select clause.
@GeneratedValue	If the values of the primary column are auto-increment, we need to use this annotation along with one of the following strategy types: **AUTO, IDENTITY, SEQUENCE, and TABLE**. In our case, we use the strategy IDENTITY, which specifies that the generated values are unique at table level, whereas the strategy AUTO implies that the generated values are unique at database level.

Hibernate Configuration Property

To use Hibernate Forward Engineering, you need to specify the hibernate.hbm2ddl.auto property in the Hibernate configuration file. The **“hibernate.hbm2ddl.auto”** property accepts the following values:

- **create**: If the value is created, Hibernate creates a new table in the database when the SessionFactory object is created. If a table exists in the database with the same name, it deletes the table, along with the data and creates a new table.
- **update**: If the value is updated, Hibernate first validates whether the table is present in the database or not. If it is present, Hibernate alters that table per the changes; if it is not present, Hibernate creates a new one.
- **validate**: If the value is validated, Hibernate only verifies whether the table is present. If the table does not exist, Hibernate throws an exception.
- **create-drop**: If the value is create-drop, Hibernate creates a new table when SessionFactory is created, performs the operation, and deletes the table when SessionFactory is destroyed. This value is used for testing the Hibernate code.
- **none**: It does not make any changes to the schema.

Example: *<property name=”hibernate.hbm2ddl.auto”>create</property>*

Specify Exact Database Dialect

- Hibernate uses the SQL syntax of the target database to create tables accordingly; therefore, it is important to specify database dialect information exactly to match the type of underlying database. Otherwise, you will get an error or an undesired behavior.
- For example, for MySQL database:

```
<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
```

Hands-On Lab

CLick here for Lab 305.3.1 - Hibernate Project Demonstration.

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

Practice Assignment

- ❑ Using Hibernate, create a Department Model with the following attributes:
  - ➢ int did
  - ➢ String name
  - ➢ String state
- ❑ Using Hibernate, create the relevant services for this Model (create, update name/state,find, delete).
- ❑ Do not forget to add your model to the configuration file (**hibernate.cfg.xml**).

If you have any technical questions while performing this practice assignment, ask your instructors for assistance.

Knowledge Check

What is the difference between ORM, JPA, and Hibernate?

Summary

Object Relational Mapping (ORM) is a concept/process of converting the data from Object Oriented Language to relational database, and vice versa. In Java, this is performed with the help of reflection and JDBC.

Java Persistence API is the one step above ORM. It has a high-level API and specification requirement so that different ORM tools can implement it and provide the flexibility to change the implementation from one ORM to another (e.g., if an application uses the JPA API and implementation is hibernate).

Let's Take A Break…

Time for
a Break

Hibernate Query Object and Hibernate Query Language

https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/Query.html

Hibernate Query Object and Hibernate Query Language

- The Hibernate Query Object is used to retrieve data from the database. Hibernate created a new language named Hibernate Query Language (HQL). The syntax is quite similar to database SQL language. The main difference is that HQL **uses class name instead of table name, and property names instead of column names**.
- Keywords, such as **SELECT, FROM**, and **WHERE,** etc. are *not* case-sensitive, but properties such as **table** and **column** names are case-sensitive in HQL.

Hibernate Query Object and Hibernate Query Language

(Continued)

Commonly supported clauses in HQL:

1. **HQL From**: HQL From is the same as a select clause in SQL. For example, `from Employee` is the same as `select * from Employee.` We can also create aliases, such as `from Employee emp` or `from Employee as emp.`
2. **HQL JOIN**: HQL supports inner JOIN, left outer JOIN, right outer JOIN, and full JOIN (e.g., `select e.name, a.city from Employee e INNER JOIN e.address a)`. In this query, Employee Class should have a variable named address. We will look into it in the example code.
3. **Aggregate Functions**: HQL supports commonly used aggregate functions such as count(*), count(distinct x), min(), max(), avg(), and sum().
4. **Expressions**: HQL supports arithmetic expressions (+, -, *, /), binary comparison operators (=, >=, <=, <>, !=, like), and logical operations (and, or, not), etc.
5. HQL supports order by and group by clauses.
6. HQL supports subqueries such as SQL queries.
7. HQL supports DDL and DML.

HQL - Query Interface

Query Interface:

- Query is an interface that presents inside **org.hibernate package.**
- A Query instance is obtained by calling ***Session.createQuery().***
- Query objects use **SQL** or **Hibernate Query Language (HQL)** strings to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally, to execute the query.
- **Hibernate provides different techniques to query database, including, TypedQuery, NamedQuery and Criteria API.** TypedQuery and NamedQuery are two additional Query sub-types.

```
Query query=session.createQuery();
or
TypedQuery query = session.createQuery();
```

Running HQL Queries

The **`Query`** interface defines two methods for running SELECT queries:

- `Query.getSingleResult` - for use when exactly one result object is expected.
- `Query.getResultList` - for general use in any other case.

The **`TypedQuery`** interface defines the following methods:

- `TypedQuery.getSingleResult` - for use when exactly one result object is expected.
- `TypedQuery.getResultList` - for general use in any other case.

In addition, the Query interface defines a method for running DELETE and UPDATE queries:

- `Query.executeUpdate` - for running only DELETE and UPDATE queries.

HQL Methods

Here are a few important Query methods, which will be used often in Hibernate implementations:

- **getSingleResult:** Execute a SELECT query that returns a single untyped result.
- **list():** Return the query results as a list. If the query contains multiple results per row, the results are returned in an instance of Object[], but use the **getResultList()** method instead of **list()** because this is a deprecated (out of date) method.
- **getResultList()** The default implementation of the getResultList() method is in org.hibernate package. Query calls the **list()** method. Execute a SELECT query and return the query results as an untyped list.

General rule for the getResultList():

- If select contains a single expression and it is an entity, the result is that entity.
- If select contains a single expression and it is a primitive, the result is that primitive.
- If select contains multiple expressions, the result is **List<Object[]>**, containing the corresponding primitives/entities.

HQL Methods (continued)

- **executeUpdate():** This method is used to run the update/delete query. It returns the number of entities updated or deleted.
- **setParameter():** Bind a value to a JDBC-style query parameter. The Hibernate type of the parameter is first detected via the usage/position in the query, and if not sufficient, secondly guessed from the class of the given object.
- **setMaxResults():** Set the maximum number of rows to retrieve. If not set, there is no limit to the number of rows retrieved.

Visit the link below for more information.

https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/Query.html#getResultList()

Clauses in the HQL

FROM Clause:

You will use the FROM clause if you want to load a complete persistent object into memory. Following is the simple syntax of using the FROM clause:

```
public static void main(String[] args) {
   SessionFactory factory = new Configuration().configure().buildSessionFactory();
   Session session = factory.openSession();
   String hql = "FROM User"; // Example of HQL to get all records of user class
   TypedQuery query = session.createQuery(hql);
   List<User> results = query.getResultList();
   for (User u : results) {
       System.out.println("User Id: " +u.getId() + "|" + " Full name:" + u.getFullname() +"|"+ "Email: " + u.getEmail()
+"|"+ "password" + u.getPassword());
   }
}
```

Clauses in the HQL (continued)

WHERE Clause: If you want to narrow the specific objects that are returned from storage, use the WHERE clause. Following is the simple syntax of using the **WHERE** clause:

```
public static void main(String[] args) {
   SessionFactory factory = new Configuration().configure().buildSessionFactory();
   Session session = factory.openSession();
   String hql = "FROM User u WHERE u.id = 2" ; // Example of HQL to get all records of user class
   TypedQuery query = session.createQuery(hql);
   List<User> results = query.getResultList();
   for (User u : results) {
       System.out.println("User Id: " +u.getId() + "|" + " Full name:" + u.getFullname() +"|"+ "Email: " + u.getEmail() +"|"+
"password" + u.getPassword());
   }}
```

HQL supports the list of expressions that are used in the where clause:

- math operators: +, -, *, /
- comparison operators: =, >=, <=, <>, !=, like
- logical operations: and, or, not
- is empty, is not empty, in, not in, between, is null, is not null, member of and not member of
- string concatenation concat(…,…)
- current_date(), current_time(), and current_timestamp()
- str() for converting values to a readable string

Clauses in the HQL (continued)

ORDER BY Clause:

To sort your HQL query results, you will need to use the ORDER BY clause. You can order the results by any property on the objects in the result set (ascending [ASC] or descending [DESC]). Following is the simple syntax of using ORDER BY clause:

```
public static void main(String[] args) {
   SessionFactory factory = new Configuration().configure().buildSessionFactory();
   Session session = factory.openSession();
   String hql = "FROM User E WHERE E.id > 3 ORDER BY E.salary DESC";
   TypedQuery query = session.createQuery(hql);
   List<User> results = query.getResultList();
   for (User u : results) {
       System.out.println("User Id: " +u.getId() + "|" + " Full name:" + u.getFullname() +"|"+ "Email: " +
u.getEmail() +"|"+ "password" + u.getPassword());
   }
}
```

Multiple SELECT Expressions

The SELECT clause may also define composite results:

```
SELECT U.salary, U.fullname FROM User AS U;
```

The result list of above query contains Object[] elements, one per result. The length of each result **Object[]** element is **2**. The first array cell contains the salary (`U.salary`) and the second array cell contains the Full name (`U.fullname`). The following code demonstrates the above query

```
public static void main(String[] args) {
   SessionFactory factory = new Configuration().configure().buildSessionFactory();
   Session session = factory.openSession();
   TypedQuery<Object[]> queryy = session.createQuery( "SELECT U.salary, U.fullname FROM User AS U", Object[].class);
List<Object[]> resultss = queryy.getResultList();
for (Object[] a : resultss) {
   System.out.println("Salary: " + a[0] + ", Full name: " + a[1]); }
}}
```

Clauses in the HQL (continued)

GROUP BY Clause and Aggregate function

This clause lets Hibernate pull information from the database and group it based on a value of an attribute; and typically, uses the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

```
public static void main(String[] args) {
   SessionFactory factory = new Configuration().configure().buildSessionFactory();
   Session session = factory.openSession();
   String hql = "SELECT SUM(U.salary), U.city FROM User U GROUP BY U.city";
                //The query returns Object[] arrays of length 2
   TypedQuery query = session.createQuery(hql);
   List<Object[]> result =query.getResultList();
   for (Object[] o : result) {
     System.out.println("Total salary " +o[0] +" | city: "+ o[1] );
    }
} }
```

Hibernate Parameterized Query

Using Named Parameters:

Hibernate supports named parameters in its HQL queries. Following is the simple syntax of using named parameters:

```
    String hql = "FROM User u WHERE u.id = :id";
    TypedQuery query = session.createQuery(hql);
    query.setParameter("id", 4);
    List<User> result = query.getResultList();
    for (User u : result) {
        System.out.println("User Id: " + u.getId() + "|" + " Full name:" + u.getFullname() +
"|" + "Email: " + u.getEmail() + "|" + "password" + u.getPassword())
    }
```

Hands - On Lab

Click here for LAB 305.3.2 - Demonstration of Hibernate Query Language HQL.

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

Problem With HQL and SQL

Problem:

A major disadvantage of having HQL and SQL scattered across data access objects is that it makes the code unreadable. Hence, it might make sense to group all HQL and SQL in one place and use only their reference in the actual data access code.

Solution:

Fortunately, Hibernate allows us to do this with **Named Queries.**

Overview of Hibernate Named Query

- **Hibernate provides "NamedQuery" that can be defined at a central location and can be used anywhere in the code.** The Hibernate named query is a way to use any query by some meaningful name. It is like using alias names, whereas the Hibernate framework provides the concept of named queries so that application programmers need not scatter queries to all the Java code.
- **We can create a NamedQuery for both HQL and Native SQL.** A named query is a statically defined query with a predefined, unchangeable query string. Named queries are compiled when SessionFactory is instantiated (essentially, when your application starts up). They are validated when the session factory is created.
- If you want to use named query in Hibernate, you need to have knowledge of **@NamedQueries** annotations and **@NamedQuery** annotations:
  - **@NameQuery** annotation is used to define the single-named query.
  - **@NameQueries** annotation is used to define the multiple-named queries.

Note: You cannot have two named queries with the same name in Hibernate. Hibernate shows *fail-fast* behavior in this regard, and will show an error in the application start-up.

Example - Hibernate Named Query

NamedQuery can be defined in Hibernate mapping files or by using Hibernate annotations.

Named query definition has two important attributes:

- **name**: The name attribute of a named query by which it will be located using hibernate session.
- **query**: Here you will give the HQL statement to get executed in the database.

User Entity Class - Hibernate Named Query @NamedQuery Annotation

```
@NamedQueries({
     @NamedQuery(  name = "findStockByStockCode", query = "from Stock s where s.stockCode = :stockCode" )
      })
             @Entity
             @Table(name = "stock")
             public class Stock implements java.io.Serializable {  . . ……….
```

Call a named query - We can call the named query via getNamedQuery method.

```
Query query = session.getNamedQuery("findStockByStockCode")
.setString("stockCode", "7277");

Query query = session.getNamedQuery("findStockByStockCodeNativeSQL")
 .setString("stockCode", "7277");
```

Example - Hibernate Named Query (continued)

```
public void UpdateEmployee()
            {   SessionFactory factory = new Configuration().configure().buildSessionFactory();
                Session session = factory.openSession();
                Transaction t = session.beginTransaction();
                TypedQuery query = session.getNamedQuery("updateEmpById");
                query.setParameter("name", "Elded Leon");
                query.setParameter("id", 7);
                int rowsAffected = query.executeUpdate();
                  if (rowsAffected > 0) {
                       System.out.println(rowsAffected + "(s) were inserted");
                   }
                   t.commit();
                   System.out.println("successfully saved");
            }
```

```
 //Using @NamedQueries for multiple  HQL
 @NamedQueries({
    @NamedQuery(name="updateEmpById", query = "update Employee set Name = :name where id = :id")
)}
```

Advantages of Named Queries

1. Fail fast: Their syntax is checked when the session factory is created, making the application fail fast in case of an error.
2. Reusable: They can be accessed and used from several places, which increase reusability.

Hands-On Lab

Click here for LAB 305.3.3- Demonstration - Named Queries in Hibernate

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

JDBC vs. Hibernate

- Hibernate performs an Object-Relational Mapping (ORM) framework, while JDBC is simply a database connectivity API.
- With JDBC, you need to write database-specific SQL statements for your DDL, DML, and queries. JPA allows you to avoid writing database-specific DDL or DML, and to express queries in terms of the Java entities rather than native SQL tables and columns.
- JDBC is database dependent (i.e., one needs to write different codes for different databases). Hibernate is database independent and the same code can work for many databases with minor changes.
- The benefits of using Hibernate over JDBC are:
  - There is reduced boilerplate code.
  - There is automatic Object Mapping.
  - It is easy to migrate to a new database.

References

https://www.vogella.com/tutorials/JavaPersistenceAPI/article.html#jpaintro_overview

https://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/objectstate.html

https://www.interviewbit.com/hibernate-interview-questions/

End of Module