# ORM

ORM stands for Object Relational Mapping. It's a technique in computer software that enables the interaction between a database management system (DBMS) and an object-oriented programming language. ORM allows developers to work with databases using high-level, object-oriented constructs, rather than writing raw SQL queries. The ORM maps objects in the program to records in the database, abstracting away the underlying DBMS and making it easier for the developer to interact with the database in a natural, intuitive way.

# JPA

* JPA (Java Persistence API) is a Java specification for accessing, persisting and managing data between Java objects/classes and a relational database. It provides a standardized and object-oriented approach to interact with databases, enabling the use of object-relational mapping (ORM) to map Java objects to database tables and manage database transactions.
* JPA specification defines a set of APIs for reading, writing and accessing data in a database. JPA provides a uniform API for database access, which makes it possible to switch between different JPA implementations (such as Hibernate, EclipseLink, and Apache OpenJPA) without changing the application code.
* With JPA, the developer can work with database records as Java objects, making it easier to work with data in a more natural, high-level way. JPA provides automatic mapping of Java classes to database tables and manages the database connections and transactions transparently to the application. JPA also supports advanced database features like caching, lazy loading, and optimistic locking.
* JPA can be used in various types of applications, such as standalone Java applications, Java EE web applications, and Java EE enterprise applications. It supports a wide range of database management systems and provides a powerful and flexible toolset for database access.

# Advantages of using the Hibernate framework

* **Database Independent:** Hibernate provides a database-agnostic API, enabling developers to switch between different databases without having to change the application code.
* **Object-Relational Mapping (ORM):** Hibernate provides a high-level, object-oriented API for persistence, making it easier for developers to work with databases.
* **Performance:** Hibernate provides various caching mechanisms that can improve application performance and reduce the load on the database.
* **Automatic Schema Generation:** Hibernate can automatically generate the database schema based on the Java classes, reducing the need for manual configuration.
* **Lazy Loading:** Hibernate supports lazy loading, which can improve performance by only loading data from the database when it is needed.
* **Robustness:** Hibernate is a robust and reliable framework, with a large community of developers and extensive documentation.
* **Flexibility:** Hibernate provides a flexible and powerful API, allowing developers to customize their persistence logic and make use of advanced database features.
* **Integration:** Hibernate integrates well with other Java frameworks, such as Java EE and Spring, making it easy to build complex, multi-tier applications.

# Hibernate Architecture

The Hibernate architecture includes many objects such as a persistent object, session factory, transaction factory, connection factory, session, transaction etc.

The Hibernate architecture is categorized into four layers.

* Java application layer
* Hibernate framework layer
* Backhand API layer
* Database layer

Let's see the diagram of hibernate architecture:

This is the high-level architecture of Hibernate with a mapping file and configuration file.

Hibernate framework uses many objects such as session factory, session, transaction etc. along with existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).



# Components of Hibernate Architecture

**SessionFactory (org.hibernate.SessionFactory):** SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We can get an instance of org. hibernate.Session using SessionFactory.

**Session (org.hibernate.Session):** Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connection and works as a factory for org. hibernate.Transaction.

**Persistent objects:** Persistent objects are short-lived, single-threaded objects that contain persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one org.hibernate.Session.

**Transient objects:** Transient objects are persistent classes instances that are not currently associated with an org.hibernate.Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed org.hibernate.Session.

**Transaction (org.hibernate.Transaction):** Transaction is a single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC or JTA transaction. An org.hibernate.A session might span multiple org. hibernate. Transaction in some cases.

**ConnectionProvider (org.hibernate.connection.ConnectionProvider):** ConnectionProvider is a factory for JDBC connections. It provides abstraction between the application and underlying javax.sql.DataSource or java.sql.DriverManager. It is not exposed to application, but it can be extended by the developer.

**TransactionFactory (org.hibernate.TransactionFactory):** A factory for org. hibernate. Transaction instances.

These components work together to provide a robust and flexible persistence layer for Java applications, making it easier for developers to work with databases in a natural, high-level way.

# Create a simple Hibernate application using XML

Here, we are going to create the first hibernate application without IDE. For creating the first hibernate application, we need to follow the following steps:

1. Create the Persistent class
2. Create the mapping file for the Persistent class
3. Create the Configuration file
4. Create the class that retrieves or stores the persistent object

### Hibernate Configuration file

The configuration file in Hibernate is an XML file or a properties file named "hibernate.cfg.xml" or "hibernate.properties", respectively. It contains configuration information for the Hibernate framework, such as the following:

1. **Database Connection:** The configuration file contains information about the database connection, such as the JDBC driver, URL, username, and password.
2. **Dialect**: The configuration file specifies the Hibernate dialect to use, which is specific to the database management system (DBMS) being used. The dialect specifies the SQL syntax and behaviour of the database.
3. **Mapping** **Resources**: The configuration file lists the mapping resources, which are the XML or annotations-based files that define the mapping between the Java classes and the database tables.
4. **Caching**: The configuration file can specify the caching settings, such as the level of caching to use and the cache provider.
5. **Transaction**: The configuration file can specify the transaction factory and transaction manager, which are used to manage transactions in the application.
6. **Properties**: The configuration file can specify additional properties, such as the show\_sql property that determines whether to log the SQL statements generated by Hibernate.

The configuration file is loaded by the Hibernate Configuration class, which uses the information in the file to create a SessionFactory object. The SessionFactory is then used to create Sessions, which are used to interact with the database.

#### **Hibernate configuration**

Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called hibernate.properties, or as an XML file named hibernate.cfg.xml.

I will consider XML formatted file hibernate.cfg.xml to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

#### **Hibernate Properties**

Following is the list of important properties, you will be required to configure for a database in a standalone situation −

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Properties** | **Description** |
| 1 | hibernate.dialect | This property makes Hibernate generate the appropriate SQL for the chosen database. |
| 2 | hibernate.connection.driver\_class | The JDBC driver class. |
| 3 | hibernate.connection.url | The JDBC URL to the database instance. |
| 4 | hibernate.connection.username | The database username. |
| 5 | hibernate.connection.password | The database password. |
| 6 | hibernate.connection.pool\_size | Limits the number of connections waiting in the Hibernate database connection pool. |
| 7 | hibernate.connection.autocommit | Allows autocommit mode to be used for the JDBC connection. |

If you are using a database along with an application server and JNDI, then you would have to configure the following properties –

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Properties** | **Description** |
| 1 | hibernate.connection.datasource | The JNDI name defined in the application server context, which you are using for the application. |
| 2 | hibernate.jndi.class | The InitialContext class for JNDI. |
| 3 | hibernate.jndi.<JNDIpropertyname> | Passes any JNDI property you like to the JNDI InitialContext. |
| 4 | hibernate.jndi.url | Provides the URL for JNDI. |
| 5 | hibernate.connection.username | The database username. |
| 6 | hibernate.connection.password | The database password. |

#### Hibernate with MySQL Database

MySQL is one of the most popular open-source database systems available today. Let us create hibernate.cfg.xml configuration file and place it in the root of your application's classpath. You will have to make sure that you have testdb database available in your MySQL database and you have a user test available to access the database.

The XML configuration file must conform to the Hibernate 3 Configuration DTD, which is available at <http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd>.

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
 <session-factory>  
 <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  
 <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernate</property>  
 <property name="hibernate.connection.username">root</property>  
 <property name="hibernate.connection.password">root</property>  
 <property name="hibernate.connection.autocommit">true</property>  
 <property name="dialect">org.hibernate.dialect.MySQLDialect</property>  
 <property name="show\_sql">true</property>  
 <property name="hibernate.hbm2ddl.auto">update</property>  
 </session-factory>  
</hibernate-configuration>

Following is the list of various important databases dialect property type −

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Database & Dialect Property** | **Dialect Property** |
| 1 | DB2 | org.hibernate.dialect.DB2Dialect |
| 2 | HSQLDB | org.hibernate.dialect.HSQLDialect |
| 3 | HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| 4 | Informix | org.hibernate.dialect.InformixDialect |
| 5 | Ingres | org.hibernate.dialect.IngresDialect |
| 6 | Interbase | org.hibernate.dialect.InterbaseDialect |
| 7 | Microsoft SQL Server 2000 | org.hibernate.dialect.SQLServerDialect |
| 8 | Microsoft SQL Server 2005 | org.hibernate.dialect.SQLServer2005Dialect |
| 9 | Microsoft SQL Server 2008 | org.hibernate.dialect.SQLServer2008Dialect |
| 10 | MySQL | org.hibernate.dialect.MySQLDialect |
| 11 | Oracle (any version) | org.hibernate.dialect.OracleDialect |
| 12 | Oracle 11g | org.hibernate.dialect.Oracle10gDialect |
| 13 | Oracle 10g | org.hibernate.dialect.Oracle10gDialect |
| 14 | Oracle 9i | org.hibernate.dialect.Oracle9iDialect |
| 15 | PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| 16 | Progress | org.hibernate.dialect.ProgressDialect |
| 17 | SAP DB | org.hibernate.dialect.SAPDBDialect |
| 18 | Sybase | org.hibernate.dialect.SybaseDialect |
| 19 | Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |

## Creating session Object using xml

In above xml we have provided all properties to create session object

1. We have to create ***SessionFactory*** object so that we can use same object to create session’s object. But ***SessionFactory*** is an Interface so we have to find implementation for this class. We are using ***Configuration*** class implementation to fetch ***SessionFactory.***

*Configuration configuration = new Configuration();*

After creating configuration object, we have to configure *hibernate.cfg.xml* file for this object.

*configuration.configure("hibernate.cfg.xml");*

Now we can fetch **SessionFactory** Object using this **configuration** object.

*SessionFactory factory = configuration.buildSessionFactory();*

Initially there is no open session so we have to open new session, like below.

*Session session = factory.openSession();*

# Create simple Hibernate application using annotation

in Java with annotations. The example demonstrates the use of annotations for mapping Java classes to database tables and for mapping class attributes to table columns.

The Employee class is annotated with **@Entity**, which indicates to Hibernate that this class should be persisted to a database. The id attribute is annotated with @Id to indicate it is the primary key and with **@GeneratedValue** to indicate that its value should be generated automatically. The name and salary attributes are not annotated, but their values will be stored in the corresponding columns of the Employee table.

The Main class is responsible for saving an employee object to the database using Hibernate. It first creates a SessionFactory instance and then opens a Session to interact with the database. The Session is used to start a transaction, save the Employee object, and commit the transaction. Finally, the SessionFactory is closed.

Note that the example assumes the existence of a configuration file named hibernate.cfg.xml, which provides Hibernate with information such as the database connection details.

|  |  |
| --- | --- |
| **Model class-**  @Entity @Table(name = "employeeUsingAnnotations") public class EmployeeUsingAnnotations {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  private int id;  private String firstName, lastName; } | Model Mapping class-  <?xml version="1.0"?> <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"> <hibernate-mapping>  <class name="model.EmployeeUsingAnnotations" table="employeeUsingAnnotations">  <id name="id">  <column name="EMPID" length="8"/>  <generator class="assigned"/>  </id>  <property name="firstName" column="FIRSTNAME" type="java.lang.String"/>  <property name="lastName" column="LASTNAME" type="java.lang.String"/>  </class> </hibernate-mapping> |
| **Hibernate config file-**  <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd"> <hibernate-configuration>  <session-factory>  <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernate</property>  <property name="hibernate.connection.username">root</property>  <property name="hibernate.connection.password">root</property>  <property name="dialect">org.hibernate.dialect.MySQLDialect</property>  <property name="show\_sql">true</property>  <property name="hibernate.hbm2ddl.auto">update</property>  <property name="hibernate.connection.autocommit">true</property>  <mapping resource="employeeUsingAnnotations.hbm.xml"/>  </session-factory> </hibernate-configuration> | |

# Hibernate configuration using java and annotations- NO USE OF XML FILE

Hibernate is being configured using Java code instead of an XML configuration file. The Configuration class is used to configure Hibernate and the StandardServiceRegistryBuilder class is used to build the SessionFactory.

1. The first step is to create a Configuration object and set the properties for the database connection. This is done using a Properties object, which is populated with key-value pairs for the properties. The properties include the driver class, URL, username, and password for the database connection.

|  |
| --- |
| Configuration configuration = new Configuration();  // Set Hibernate properties Properties properties = new Properties(); properties.put(Environment.*DRIVER*, "com.mysql.jdbc.Driver"); properties.put(Environment.*URL*, "jdbc:mysql://localhost:3306/test\_db"); properties.put(Environment.*USER*, "user"); properties.put(Environment.*PASS*, "password"); properties.put(Environment.*DIALECT*, "org.hibernate.dialect.MySQL5Dialect"); properties.put(Environment.*SHOW\_SQL*, "true"); properties.put(Environment.*HBM2DDL\_AUTO*, "create-drop"); configuration.setProperties(properties); |

1. Next, the Configuration object is used to add the annotated classes to be persisted by Hibernate. In this example, the Employee class is added using the addAnnotatedClass method.

|  |
| --- |
| // Add annotated classes configuration.addAnnotatedClass(Employee.class); |

Finally, the SessionFactory is built using the StandardServiceRegistryBuilder class and the build method of the Configuration object. The SessionFactory is the central point of the Hibernate configuration and is used to create Session objects for interacting with the database.

|  |
| --- |
| // Build session factory StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder()  .applySettings(configuration.getProperties()); sessionFactory = configuration.buildSessionFactory(builder.build()); |

A more complete configuration would typically include additional properties such as the dialect, show SQL, and other Hibernate-specific properties.

# Generator Classes

Generator classes in Hibernate are used to generate unique identifier values for entities. The identifier is used to represent the entity's primary key in the database and is usually assigned automatically by Hibernate.

Below are the 6 built-in generator classes in Hibernate:

1. assigned - the developer is responsible for assigning identifier values. The value can be specified either in the entity class or through the use of the @Id annotation.

Example:

@Entity

public class Employee {

@Id

private Long id;

// other fields and methods

}

1. increment - uses a hi/lo algorithm to efficiently generate unique numbers. This algorithm maintains a high and low value to generate identifier values.

Example:

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.INCREMENT)

private Long id;

// other fields and methods

}

1. sequence - uses a database sequence. In this case, a sequence is created in the database and its values are used to generate identifier values.

Example:

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private Long id;

// other fields and methods

}

1. identity - uses an identity column. An identity column is a column that automatically generates a unique value for each new row in the table.

Example:

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

// other fields and methods

}

1. table - uses a separate database table to store identifier values. In this case, a table is created in the database and its values are used to generate identifier values.

Example:

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.TABLE)

private Long id;

// other fields and methods

}

1. uuid.hex - uses a universally unique identifier in hexadecimal format. A universally unique identifier (UUID) is a unique identifier that is guaranteed to be unique across time and space.

Example:

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.UUID)

private String id;

// other fields and methods

}

Note: It is important to choose the appropriate generator class based on the database being used, the performance requirements, and the type of identifier value desired.

## What is hi/lo algorithm

The hi/lo algorithm is a technique used by Hibernate to generate unique identifier values for entities in an efficient and scalable manner.

The algorithm works by maintaining a high value and a low value. The low value is used to generate the identifier values for a batch of entities, and the high value is used to determine the range of values to be used for each batch. When the low value reaches the high value, the high value is updated, and a new batch of values is generated.

This algorithm helps to reduce the number of database round trips required to generate unique identifier values, and it also helps to ensure that identifier values are unique even in a concurrent environment.

## Depreciated generator classes hibernate

1. UUIDHexGenerator - This class generated universally unique identifiers (UUIDs) in hexadecimal format. The recommended alternative is to use the UUIDGenerator class, which generates UUIDs as a string representation of the standard UUID format.
2. IncrementGenerator - This class generated identifier values using a database-specific auto-increment mechanism. The recommended alternative is to use a database-specific identifier generator, such as IDENTITY for auto-incrementing identifiers or SEQUENCE for using a database sequence.
3. SelectGenerator - This class generated identifier values using a database-specific mechanism, such as a select statement, to obtain the next value. The recommended alternative is to use a database-specific identifier generator, such as SEQUENCE for using a database sequence.
4. SequenceGenerator - This class generated identifier values using a database-specific sequence. The recommended alternative is to use the SequenceStyleGenerator class, which supports multiple database types and allows for more fine-grained configuration of the sequence generation process.

# SQL Dialects

SQL Dialects in Hibernate refers to a set of classes that translate Hibernate’s internal representation of SQL statements into the syntax of a specific database management system (DBMS). These dialect classes tell Hibernate which SQL keywords to use for various operations and how to perform database-specific tasks, such as using sequences for generating primary keys.

Here is a list of some of the supported SQL Dialects in Hibernate:

1. H2
2. MySQL
3. Oracle
4. PostgreSQL
5. Microsoft SQL Server
6. SQLite
7. Sybase SQL Anywhere
8. HSQLDB
9. DB2
10. Informix

To specify the SQL Dialect in Hibernate, it is specified in the hibernate.cfg.xml configuration file or in the properties file used to create the SessionFactory.

# Commonly used annotation in Hibernate

Here is a list of some of the commonly used annotations in Hibernate along with their usage:

These are Java annotations commonly used in JPA (Java Persistence API) for mapping Java objects to database tables.

Here is a comprehensive list of annotations used in Hibernate, along with their purpose and category:

1. @Entity - Marks a class as a persistent entity. Category: Class-level
2. @Id - Specifies the primary key of an entity. Category: Property-level
3. @GeneratedValue - Specifies how the primary key is generated. Category: Property-level
4. @Column - Maps a property to a column in the database table. Category: Property-level
5. @OneToOne - Specifies a one-to-one relationship between two entities. Category: Property-level
6. @OneToMany - Specifies a one-to-many relationship between two entities. Category: Property-level
7. @ManyToOne - Specifies a many-to-one relationship between two entities. Category: Property-level
8. @ManyToMany - Specifies a many-to-many relationship between two entities. Category: Property-level
9. @JoinColumn - Specifies the name of the foreign key column. Category: Property-level
10. @Table - Specifies the name of the database table associated with an entity. Category: Class-level
11. @SecondaryTable - Specifies a secondary table for an entity. Category: Class-level
12. @Temporal - Specifies the mapping of a date/time property. Category: Property-level
13. @Transient - Specifies that a property should not be persisted. Category: Property-level
14. @Lob - Specifies the mapping of a Large Object (LOB) property. Category: Property-level
15. @Enumerated - Specifies the mapping of an enumerated type property. Category: Property-level
16. @Basic - Specifies a basic property mapping. Category: Property-level
17. @Embedded - Specifies an embedded object. Category: Property-level
18. @Embeddable - Specifies a class whose properties are to be included as part of the properties of another class. Category: Class-level
19. @AttributeOverride - Overrides the mapping of a property inherited from an embedded class. Category: Property-level
20. @AttributeOverrides - Overrides the mappings of multiple properties inherited from an embedded class. Category: Property-level
21. @ElementCollection - Specifies a collection of basic or embeddable objects. Category: Property-level
22. @CollectionTable - Specifies the table for storing the elements of a collection. Category: Property-level
23. @JoinTable - Specifies the join table for a many-to-many relationship. Category: Property-level
24. @MapKeyColumn - Specifies the column for the map key of a map. Category: Property-level
25. @MapKey - Specifies the map key for a map. Category: Property-level
26. @MapKeyEnumerated - Specifies the mapping of the map key for an enumerated type. Category: Property-level
27. @MapKeyTemporal - Specifies the mapping of the map key for a date/time type. Category: Property-level
28. @MapKeyJoinColumn - Specifies the join column for the map key of a map. Category: Property-level
29. @MapKeyJoinColumns - Specifies multiple join columns for the map key of a map. Category: Property-level

# Inheritance Mapping

Hibernate is an Object-Relational Mapping (ORM) framework used in Java to map objects to relational database tables. Inheritance mapping in Hibernate allows you to map the inheritance hierarchy of your object-oriented domain model to a relational database.

Here are the main inheritance mapping strategies in Hibernate:

1. Single Table per Class Hierarchy
2. Table per Subclass
3. Table per Concrete Class

## Single table per class hierarchy

You need to use @Inheritance(strategy=InheritanceType.SINGLE\_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as discriminator column) is created in the table to identify the class.

Table per Hierarchy (TPH) is a strategy for mapping inheritance in object-relational mapping (ORM) frameworks such as JPA (Java Persistence API) or Hibernate. In TPH, a single table is used to store all the entities in a class hierarchy. The table contains columns for the properties of each entity in the hierarchy, as well as a column to distinguish which type of entity each row represents.

For example, if there is an inheritance hierarchy with a base class "Animal" and two subclasses "Cat" and "Dog", a single table would be created to store all instances of "Animal", "Cat", and "Dog". The table would contain columns for the properties of "Animal", "Cat", and "Dog", and a column to distinguish which type of entity each row represents.

TPH is simple to implement and has good performance, but it can be inflexible when the inheritance hierarchy is changed or if entities are added or removed from the hierarchy. TPH is best suited for simple inheritance hierarchies where the changes are unlikely.

|  |  |
| --- | --- |
| **Animal class- parent class**  @Entity @Inheritance(strategy=InheritanceType.*SINGLE\_TABLE*) @DiscriminatorColumn(name="animal\_type") public class Animal {  @Id  @GeneratedValue  private long id;  private String name;  // other properties and methods  } | **Cat class- Child class**  @Entity @DiscriminatorValue("Cat") public class Cat extends Animal {  private int numberOfLives;  // other properties and methods  } |
| **Dog class- Child class**  @Entity @DiscriminatorValue("Dog") public class Dog extends Animal {  private String breed;  // other properties and methods  } | |

## Table per subclass

As we have specified earlier, in case of table per subclass strategy, tables are created as per persistent classes but they are treated using primary and foreign key. So, there will not be any duplicate column in the relation.

We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and @PrimaryKeyJoinColumn annotation in the subclasses.

Table per subclass is a design pattern in database design where each class in a class hierarchy is represented as a separate table, with a foreign key to the parent table. This approach can provide more efficient storage, as it eliminates redundant data, but it also increases the complexity of querying the data.

In this design pattern, each subclass has its own table, and a foreign key to the parent class. The parent class table includes columns to represent properties common to all subclasses, while each subclass table includes columns to represent properties specific to that subclass.

\* Example: A class hierarchy of animals, where each class has its own table, connected by a foreign key to the parent table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table: Animals   | Anima lid | AnimalType | AnimalName | AnimalWeight | | --- | --- | --- | --- | | 1 | Mammal | Lion | 200 | | 2 | Bird | Eagle | 10 | | 3 | Fish | Salmon | 5 | | Table: Birds   | AnimalID | AnimalNumberOfWings | AnimalCanFly | | --- | --- | --- | | 2 | 2 | True | |
| Table: Fish   | AnimalID | AnimalGills | AnimalLivesInWater | | --- | --- | --- | | 3 | True | True | | Table: Mammals   | AnimalID | AnimalNumberOfLegs | AnimalWarmBlooded | | --- | --- | --- | | 1 | 4 | True | |

In this example, the "Animals" table includes columns to represent properties common to all subclasses, while each subclass table includes columns to represent properties specific to that subclass. The foreign key "AnimalID" in each subclass table links it to the parent "Animals" table.

The advantage of this design pattern is that it eliminates redundant data and provides more efficient storage. The disadvantage is that it can make querying the data more complex, as data from multiple tables must be joined to retrieve information about a single object.

**Ex-**

|  |  |
| --- | --- |
| **Animal class**  @Entity @Inheritance(strategy=InheritanceType.*JOINED*) @Table(name="Animals") public class Animal {   @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  private Long id;  private String type;  private String name;  private int weight;  // Getters and Setters  } | **Child class- Bird**  @Entity @Table(name = "Birds") public class Bird extends Animal {   private int numberOfWings;  private boolean canFly;  // Getters and Setters  } |
| **Child class- Fish**  @Entity @Table(name = "Fish") public class Fish extends Animal {   private boolean gills;  private boolean livesInWater;  } | **Child class- Mammal**  @Entity @Table(name="Mammals") public class Mammal extends Animal {  private int numberOfLegs;  private boolean warmBlooded;  // Getters and Setters  } |

In this example, we use JPA (Java Persistence API) annotations to define the Animal, Mammal, Bird, and Fish classes, with each subclass having its own table, connected by a foreign key to the parent Animal table. The @Inheritance annotation specifies that the inheritance strategy is JOINED, meaning that each subclass has its own table. The advantage of using this design pattern is that it eliminates redundant data and provides more efficient storage. The disadvantage is that it can make querying the data more complex, as data from multiple tables must be joined to retrieve information about a single object.

## Table Per Concrete class

Table Per Concrete Class (TPC) is a database design strategy where each concrete class in a class hierarchy is mapped to its own separate table in the database. This means that each concrete class has its own table that includes all of its properties and attributes, including those inherited from its parent classes.

This strategy is used when each concrete class has its own unique attributes and there is no need for a shared or common table for the entire hierarchy.

Advantages of TPC include reduced data redundancy, improved data retrieval performance, and easier database management. However, it can also lead to difficulties in enforcing referential integrity constraints and increased complexity in querying data across the class hierarchy.

In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

**@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)** specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

In the Table Per Concrete Class inheritance strategy, each concrete class has its own table in the database, and the table only contains the columns for the attributes declared in the concrete class. Since each concrete class has its own table, there is no need to override any inherited attributes, so @AttributeOverrides is not required in this strategy.

**Ex-**

|  |  |
| --- | --- |
| **Parent Class- Person.java**  @Entity @Inheritance(strategy = InheritanceType.*TABLE\_PER\_CLASS*) public class Person {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int pid;  String pname;  //getters and setters  } | **Child class- Student.java**  @Entity public class Student extends Person {   int studentId;  String studentName;  //getters and setters  } |
| **Child class- Teacher.java**  @Entity public class Teacher extends Person {   int teacherId;  String teacherName;  //getters and setters  } | |

## Advantages and disadvantages of all Inheritance mapping in Hibernate

### Single table per class hierarchy:

#### Advantages:

* Simplest and most straightforward design.
* Queries can be optimized for inheritance structures.
* Good for simple inheritance structures.

#### Disadvantages:

* Can result in a large and complex table.
* Null values are often used to represent missing information in subclasses.
* Polymorphic queries can be complex.

### Table per subclass:

#### Advantages:

* Each subclass gets its own table, reducing complexity.
* Subclasses can have their own specific columns.
* Avoids the need for null values.

#### Disadvantages:

* Joining tables can be complex and slow.
* More tables to manage and maintain.
* Data redundancy can lead to inconsistencies.

### Table per concrete class:

#### Advantages:

* Avoids complex joins.
* No data redundancy.
* Efficient storage of data specific to each concrete class.

#### Disadvantages:

* Can lead to a large number of tables.
* Not suitable for complex inheritance structures.
* Adds overhead for managing multiple tables.

## Difference between Single table per class hierarchy , Table per subclass and Table per concrete class

**Single Table per Class Hierarchy:**

All classes in the hierarchy are stored in a single table, with columns for all properties in the hierarchy.

Pros: Simplest to implement, efficient for small hierarchy.

Cons: Columns for unused properties in some rows, inefficient for large hierarchy.

**Table per Subclass:**

Each class in the hierarchy has its own table, with a foreign key to the parent table.

Pros: Avoids columns for unused properties, efficient for large hierarchy.

Cons: More tables, more complex to implement, more joins required for queries.

**Table per Concrete Class:**

Each concrete (i.e. non-abstract) class in the hierarchy has its own table, without a foreign key to the parent.

Pros: Avoids columns for unused properties, efficient for large hierarchy with many concrete classes.

Cons: More tables, more complex to implement, duplicated data for shared properties, no direct way to query the hierarchy.

# Hibernate relationship mapping

Relationship mapping in Hibernate refers to the process of defining the relationships between entities in a Java application and tables in a relational database. This mapping allows the entities in the Java application to interact with each other as objects, while still being stored and retrieved from the database as tables.

Hibernate provides several types of relationship mapping, including:

1. **One-to-One Mapping** - This type of mapping defines a relationship between two entities where one instance of an entity corresponds to one instance of another entity. For example, a person may have a unique passport.
2. **One-to-Many Mapping** - This type of mapping defines a relationship between two entities where one instance of an entity corresponds to multiple instances of another entity. For example, one person may have multiple phones.
3. **Many-to-One Mapping** - This type of mapping defines a relationship between two entities where multiple instances of one entity correspond to one instance of another entity. For example, multiple phones may belong to one person.
4. **Many-to-Many Mapping** - This type of mapping defines a relationship between two entities where multiple instances of one entity correspond to multiple instances of another entity. For example, multiple persons may have multiple phones.

The relationship mapping information can be defined using annotations or XML mapping files. The annotations used in relationship mapping include @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany. These annotations define the relationship type, the mapping between the columns in the database and the properties in the Java entities, and the cascading of operations from the parent to the child entities.

For example, the @OneToMany annotation would define a one-to-many relationship between two entities, and would specify the mapping between the columns in the database and the properties in the Java entities. The cascading of operations, such as saving or deleting, can also be defined using this annotation.

## One-to-One Mapping

Hibernate One-to-One mapping is a relationship between two entities where one entity has a unique association with another entity. In this type of mapping, one entity is the parent and the other is the child. To implement a One-to-One mapping in Hibernate, both entities must be annotated with JPA annotations such as **@Entity** and **@Table**. The **@OneToOne** annotation is used on both entities to define the relationship and the @JoinColumn attribute is used to specify the foreign key. The owner side of the relationship can be specified using the **mappedBy** attribute. With these annotations, Hibernate can handle the persistence of these entities, automatically saving and updating them when necessary.

In hibernate, there are **primarily 3 ways to create one-to-one relationship**s between two entities. Either way, we have to use [@OneToOne](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/onetoone) annotation.

* The first technique is widely used and uses a **foreign key column** in one of the tables.
* The second technique uses a rather known solution of having a **join table** to store the mapping between the first two tables.
* The third technique is something new that uses a **common primary key** in both tables.

### foreign-key-association-one-to-one-2460824Using a Foreign Key Association

In this kind of association, a foreign key column is created in the owner entity. For example, if we have made EmployeeEntity owner, then an extra column "ACCOUNT\_ID" will be created in the Employee table. This column will store the foreign key for the Account table.

The table structure will be like this:

The join column is declared with the [@JoinColumn](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/joincolumn) annotation that looks like the [@Column](https://jakarta.ee/specifications/persistence/3.1/apidocs/jakarta.persistence/jakarta/persistence/column) annotation. It has one more parameter named *referencedColumnName*. This parameter declares the column name in the targeted entity that will be used to join.

**If no**@JoinColumn**is declared on the owner side, the defaults apply.** A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, \_ (underscore), and the name of the primary key column(s) in the owned side.

In a bidirectional relationship, one of the sides (and only one) has to be the owner. The owner is responsible for the association column(s) update. To declare any side as not responsible for the relationship, the attribute mappedBy is used. The ‘*mappedBy*‘ refers to the property name of the association on the owner’s side.

The "mappedBy” attribute declares that it is dependent on the owner entity for mapping.

|  |  |
| --- | --- |
| @Entity @Table(name = "EMPLOYEE") public class EmployeeEntity implements Serializable {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  @Column(name = "ID")  private Integer employeeId;  @OneToOne  @JoinColumn(name="ACCOUNT\_ID")  private AccountEntity account;  //Other fields, getters, and setters are hidden for brevity } | @Entity @Table(name = "ACCOUNT") public class AccountEntity implements Serializable {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  @Column(name = "ID")  private Integer accountId;  @OneToOne(mappedBy = "account")  private EmployeeEntity employee;  //Other fields, getters, and setters are hidden for brevity } |

### Using a Join Table

hibernate will create a new table that will store the primary key values from both entities. Let us start with the targeted DB structure in this technique.

In this technique, the main annotation to be used is @JoinTable. **This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables**.

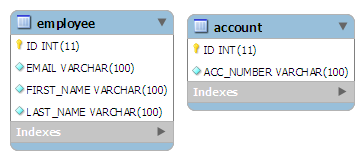
In this technique, the main annotation to be used is @JoinTable. **This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables**. Let’s see how it is used:

|  |  |
| --- | --- |
| @Entity @Table(name = "users") public class User {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  @OneToOne(mappedBy = "user", cascade = CascadeType.ALL)  private UserProfile userProfile;   // Getters and setters }} | @Entity @Table(name = "user\_profiles") public class UserProfile {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String address;  @OneToOne  @JoinTable(name = "user\_profiles\_users",  joinColumns = @JoinColumn(name = "user\_profile\_id"),  inverseJoinColumns = @JoinColumn(name = "user\_id"))  private User user;  // Getters and setters }} |

In this example, the User and UserProfile classes are defined as entities, and they are mapped to separate tables in the database. The User class has a one-to-one relationship with the UserProfile class, and the relationship is defined using the @OneToOne annotation. The join table is defined using the @JoinTable annotation, which specifies the name of the join table and the columns used to join the two entities.

The mappedBy attribute in the User class defines the owner side of the relationship, while the @JoinTable annotation in the UserProfile class defines the non-owner side of the relationship. The CascadeType.ALL setting in the User class ensures that all operations performed on the User entity are cascaded to the associated UserProfile entity.

### Using a Shared Primary Key

In this technique, hibernate will ensure that it will use a common primary key value in both tables. This way primary key of EmployeeEntity can safely be assumed as the primary key of AccountEntity also.

The table structure will be like this:

In this approach, @PrimaryKeyJoinColumn is the main annotation to be used. Let us see how to use it.

|  |  |
| --- | --- |
| @Entity public class Employee {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int empId;  String empName;  @OneToOne(cascade = CascadeType.*ALL*)  @PrimaryKeyJoinColumn  Address address;  } | @Entity public class Address {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int addressId;  String empAddress;  @OneToOne(mappedBy = "address", cascade = CascadeType.*ALL*)  Employee employee;  } |

## One-to-Many Mapping

**Hibernate one-to-many mapping** is made between two entities where the first entity can have a relation with multiple instances of the second entity but the second can be associated with only one instance of the first entity. It is a **1-to-N** relationship.

For example, in any company, an employee can register for multiple bank accounts but one bank account will be associated with one and only one employee. In this hibernate one-to-many mapping annotation example, we will learn to make such mappings in the database using hibernate.

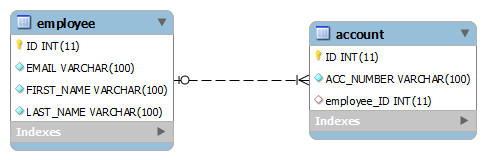
We should use one-to-many mapping to create a 1..N relationship between entities or objects.

For example, as discussed above, we have to write two entities i.e. EmployeeEntity and AccountEntity such that multiple accounts can be associated with a single employee, but one single account cannot be shared between two or more employees.

This problem can be solved in two different ways.

* One is to have a foreign key column in the ACCOUNT table i.e. EMPLOYEE\_ID. This column will refer to the primary key of the Employee table. This way no two accounts can be associated with multiple employees. The account number needs to be unique for enforcing this restriction.
* The second approach is to have a link table. Let’s say the table name is EMPLOYEE\_ACCOUNT. This table will have two columns i.e. EMP\_ID which will be a foreign key referring to the primary key in the EMPLOYEE table and similarly ACCOUNT\_ID which will be a foreign key referring to the primary key of the ACCOUNT table.

### Using Foreign Key Association

We are designing a unidirectional relationship where when we delete an employee the account is deleted as well. But when we delete an account (one of many) the employee is unaffected.

Ex-

|  |  |
| --- | --- |
| @Entity public class Person {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int personId;  String name;  @OneToMany(cascade = CascadeType.*ALL*)  @JoinColumn(name = "personId")  List<IdentificationDocument> documentList;  } | @Entity public class IdentificationDocument {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int documentId;  String documentName;  } |

### Using Link Table

This approach uses the **@JoinTable** annotation to **create a link table that stores the associations** between account and employee entities.

|  |  |
| --- | --- |
| @Entity **public class** IdentificationDocument {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  **int** documentId;  String documentName;  } | @Entity **public class** Person {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  **int** personId;  String name;  @OneToMany(cascade = CascadeType.*ALL*)  @JoinTable(name = "personId\_docId")  List<IdentificationDocument> documentList;  } |

## Many-to-One Mapping

Many-to-One Mapping is a type of relationship that exists between two entities in a database, where multiple records in one entity can be related to a single record in another entity. In this mapping, each record in the "many" entity is associated with only one record in the "one" entity. For example, consider a database of employees and departments. Each employee belongs to one department, but each department can have many employees. This represents a many-to-one relationship, where many employees are mapped to a single department. This type of mapping is often implemented using a foreign key, where the primary key of the "one" entity is stored as a foreign key in the "many" entity.

**Example-**

The Java example provided is demonstrating how to implement a many-to-one mapping using JPA (Java Persistence API) and hibernate.

In the example, there are two entities: **Employee** and **Department**. The **Employee** entity has a many-to-one relationship with the **Department** entity, meaning that many employees can be associated with a single department.

The relationship between the two entities is defined in the **Employee** entity, where there is a field called **department** annotated with **@ManyToOne**. This annotation indicates that each **Employee** can be associated with a single **Department**.

The foreign key that links the **Employee** table to the **Department** table is specified with the **@JoinColumn** annotation, which specifies the name of the column in the **Employee** table that will store the foreign key to the **Department** table.

When the code is executed, hibernate will generate the necessary SQL code to create two tables in the database: **Employee** and **Department**. The **department\_id** column in the **Employee** table will be the foreign key that links the two tables. This allows Hibernate to perform operations on the database, such as inserting, updating, and retrieving data, while maintaining the relationship between employees and departments.

|  |  |
| --- | --- |
| @Entity public class Employee {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  private Long id;   private String name;   @ManyToOne  @JoinColumn(name = "department\_id")  private Department department;   // getters and setters omitted for brevity  } | @Entity public class Department {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  private Long id;   private String name;   // getters and setters omitted for brevity  } |

## Many-to-Many Mapping

In Hibernate, a many-to-many mapping is a relationship between two entities where one entity can have multiple instances of another entity, and vice versa. This can be achieved through the use of a junction table, which acts as an intermediary between the two entities. The junction table contains the primary keys of both entities as foreign keys. The mapping is typically done using the @ManyToMany annotation in the entity class. The JoinTable annotation is used to specify the junction table, and the mappedBy attribute is used to specify the entity that is on the owning side of the relationship.

The @ManyToMany association **requires a link table that joins two entities**. Note that @ManyToMany can be either unidirectional or bidirectional.

To demonstrate many-to-many mapping using hibernate annotations, we will associate two entities i.e., ReaderEntity and SubscriptionEntity. Their database schema is given in the image. Using these tables, any application can save multiple associations between readers and subscriptions.

### Unidirectional

This is the preferred approach in most cases. We should apply *@ManyToMany* annotation only on the owning side of the relationship.

|  |  |
| --- | --- |
| @Entity public class Subscriber {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int subscriberId;   String subscriberName;  @ManyToMany(cascade = {CascadeType.*PERSIST*, CascadeType.*MERGE*})  @JoinColumn(name = "subscriber\_subscription")  List<Subscription> subscriptions;  } | @Entity public class Subscription {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int subscriptionId;  String subscriptionName;  } |

### Bidirectional

A bidirectional @ManyToMany association has an owning and a mappedBy side. We can map many to many relations either using a list, set, bag, map etc. Here, we are going to use a list for many-to-many mapping. In such a case, three tables will be created.

|  |  |
| --- | --- |
| @Entity public class Answer {   @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  private int id;  private String answername;  private String postedBy;  } | @Entity @Table(name="question") public class Question {  @Id  @GeneratedValue(strategy=GenerationType.*AUTO*)  private int id;  private String qname;   @ManyToMany(targetEntity = Answer.class, cascade = { CascadeType.*ALL* })  @JoinTable(name = "question\_answer",  joinColumns = { @JoinColumn(name = "q\_id") },  inverseJoinColumns = { @JoinColumn(name = "ans\_id") })  private List<Answer> answers;  } |

### Owning Side Entity

The relationship-owning entity is the entity that is responsible make making the association and maintaining it. In our case, I am making ReaderEntity the owner entity. @JoinTable annotation has been used to make this association.

The link table is controlled by the owning side. When an entity is removed from the @ManyToMany collection, hibernate simply deletes the joining record in the link table.

Avoid using *CascadeType.REMOVE* because it will propagate beyond the link table. Since the mapped entities (*to-be-deleted*) might be referenced by other entities on the parent side, the automatic removal might end up in a ConstraintViolationException.

So, the good practice is just to remove the parent record and hibernate will remove the associated link records.

|  |
| --- |
| **ReaderEntity.java**  @Entity(name = "ReaderEntity")  @Table(name = "READER", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID"),  @UniqueConstraint(columnNames = "EMAIL")})  public class ReaderEntity implements Serializable {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer readerId;  @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})  @JoinTable(name = "READER\_SUBSCRIPTIONS", joinColumns =  {@JoinColumn(referencedColumnName = "ID")}  , inverseJoinColumns = {@JoinColumn(referencedColumnName = "ID")})  private Set<SubscriptionEntity> subscriptions;  //Other fields, getters, setters are hidden for brevity  } |

### Mapped Entity

Our mapped entity is SubscriptionEntity which is mapped to ReaderEntity using “mappedBy” attribute. It is done in the bi-directional association.

|  |
| --- |
| **SubscriptionEntity.java**  @Entity(  @Table(name = "SUBSCRIPTION", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class SubscriptionEntity implements Serializable  {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  @Column(name = "ID", unique = true, nullable = false)  private Integer subscriptionId;  @Column(name = "SUBS\_NAME", unique = true, nullable = false, length = 100)  private String subscriptionName;    @ManyToMany(mappedBy = "subscriptions")  private Set<ReaderEntity> readers;  //Other fields, getters, and setters are hidden for brevity  } |

# Hibernate – Mapping Date, Time and Timestamp

## Mapping Java Types to JDBC Types

In SQL, we have primarily 3 types to store date-time information:

1. DATE
2. TIME
3. TIMESTAMP

There may be a few more types based on the database support.

1. TIME\_WITH\_TIMEZONE
2. TIMESTAMP\_WITH\_TIMEZONE
3. TIMESTAMP\_UTC
4. INTERVAL\_SECOND

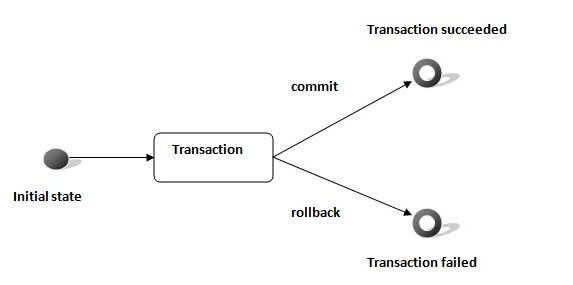
On the other hand, since [Java 8](https://howtodoinjava.com/java-8-tutorial/), we have the following java.time classes that can be mapped directly to SQL types:

Hibernate recognizes these types and we only need to provide @Column or @Basic annotation in the field. Hibernate takes care of the choosing correct data types to store/update the values in these fields.

|  |  |
| --- | --- |
| **Java Type** | **JDBC Type** |
| LocalTime | TIME |
| OffsetTime | TIME or TIME\_WITH\_TIMEZONE |
| LocalDate | DATE |
| LocalDateTime | TIMESTAMP |
| ZonedDateTime | TIMESTAMP or TIMESTAMP\_WITH\_TIMEZONE |
| OffsetDateTime | TIMESTAMP or TIMESTAMP\_WITH\_TIMEZONE |
| Instant | TIMESTAMP\_UTC |
| Duration | INTERVAL\_SECOND or NUMERIC |
| TimeZone | VARCHAR |
| ZoneOffset | VARCHAR |

|  |  |
| --- | --- |
| @Entity @AllArgsConstructor @Getter @Setter @NoArgsConstructor public class Book {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  @Column(name = "id", nullable = false)  private Long id;  @Column  private String title;  @Column  private String content;  @Column  private LocalDate lastUpdateDate;  @Column  private LocalTime lastUpdateTime;  @Column  private LocalDateTime publishedTimestamp; } | public class App {  public static void main(String[] args) {  DummyUserProvider provider = new DummyUserProvider();  Book book = new Book();  book.setTitle("sdfsdf");  book.setContent("sdfsdf");  LocalDate localDate = LocalDate.*now*();  LocalTime localTime = LocalTime.*now*();  LocalDateTime localDateTime = LocalDateTime.*now*();  book.setPublishedTimestamp(localDateTime);  book.setLastUpdateDate(localDate);  book.setLastUpdateTime(localTime);  ObjectDAO.*saveObject*(book);   }  } |

# Hibernate Transaction Management

* Hibernate Transaction Management is the process of managing database transactions in a Hibernate-based application. It is responsible for ensuring that a series of database operations are executed as a single, atomic unit of work. If any part of the transaction fails, the entire transaction is rolled back, and the database state is restored to its previous state.
* Transactions in Hibernate are managed through the org.hibernate.Transaction interface. To begin a transaction, you use the current Session's beginTransaction() method. Then, you execute the necessary database operations, and finally, you call the transaction's commit() method to persist the changes to the database. If any exceptions occur during the transaction, you can call the transaction's rollback() method to undo any changes made during the transaction.
* It is important to manage transactions properly to ensure data integrity and consistency in the database. Improper transaction management can lead to data corruption, inconsistent data, and other issues.
* In Hibernate, transactions can be managed using either programmatic transaction management or declarative transaction management. In programmatic transaction management, you manually begin, commit, and rollback transactions within your code. In declarative transaction management, you use configuration files or annotations to specify transaction boundaries and let the framework handle the transaction management.
* In summary, Hibernate Transaction Management provides a convenient and flexible way to manage database transactions in a Hibernate-based application, ensuring data integrity and consistency.

### Transaction Interface in Hibernate

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC). A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

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)** it 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 commited successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

|  |  |
| --- | --- |
| Session session = sessionFactory.openSession();  Transaction transaction = null;  try {  transaction = session.beginTransaction();  // perform database operations  transaction.commit();  } catch (Exception e) {  if (transaction != null) {  transaction.rollback();  }  throw e;  } finally {  session.close();  } | @Transactional  public void performTransaction() {  // perform database operations  } |

In this example, the @Transactional annotation on the method performTransaction indicates that a transaction should be created before the method is executed and committed after it's done. If an exception is thrown, the transaction will be automatically rolled back.

# Embeddeing object

In Hibernate, an "embedded" object is a component that is stored as part of another object's state. When you persist an object that contains an embedded object, hibernate stores the properties of both objects in a single table in the database, instead of creating a separate table for each. This can make it easier to manage relationships between objects, and can help to avoid the overhead associated with maintaining separate tables. To use embedding in Hibernate, you need to define the embedded object as a component and map it to the appropriate columns in the parent object's table.

|  |  |
| --- | --- |
| @Getter @Setter @AllArgsConstructor @NoArgsConstructor @Embeddable public class Certificate {   int cid;  String cname; } | @Entity @Getter@Setter @AllArgsConstructor @NoArgsConstructor public class Student {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int sid;  String name;  Certificate certificate; } |

# Fetch Techniques

In Hibernate, there are two main fetching techniques: eager fetching and lazy fetching.

* Eager fetching: Eager fetching loads all of an object's associations at the same time as the object itself. This can result in a large amount of data being loaded from the database, which can affect performance.
* Lazy fetching: Lazy fetching only loads an object's associations when they are actually needed. This can improve performance by reducing the amount of data that is loaded from the database.

In addition to these two techniques, hibernate also provides several advanced fetching strategies, such as batch fetching and subselect fetching, which can help you to optimize performance and manage data retrieval more effectively.

## Eager fetching

Eager fetching is a fetching technique in Hibernate where an object and its associated objects are fetched from the database at the same time. This means that all data related to the object is loaded into memory when the object is first retrieved, regardless of whether or not it is actually needed.

Eager fetching can be useful when the associations between objects are known and needed for the majority of use cases. However, it can result in a large amount of data being loaded from the database, which can negatively impact performance. Therefore, it should be used with caution, especially in applications with large amounts of data.

In Hibernate, eager fetching can be specified in the mapping file or by using annotations in the Java code.

|  |  |
| --- | --- |
| @Entity @Getter @Setter @AllArgsConstructor @NoArgsConstructor @ToString public class Department {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int dId;  String dName; } | @Entity  @Getter@Setter @AllArgsConstructor@NoArgsConstructor @ToString public class Employee {  @Id  @GeneratedValue(strategy = GenerationType.*AUTO*)  int eId;  String eName;  @ManyToOne(cascade = CascadeType.*ALL*, fetch = FetchType.*EAGER*)  Department department; } |

## Lazy fetching

Lazy fetching is a fetching technique in Hibernate where an object's associated objects are not fetched from the database until they are actually needed. This means that only the data that is required for a specific use case is loaded into memory, reducing the amount of data that needs to be retrieved from the database.

Lazy fetching can improve performance by reducing the amount of data that is loaded from the database, especially in applications with large amounts of data. However, it can also result in increased complexity when it comes to managing the associations between objects, as the data may need to be fetched from the database at a later time.

In Hibernate, lazy fetching can be specified in the mapping file or by using annotations in the Java code. When lazy fetching is enabled, hibernate will only retrieve the associated objects when they are accessed for the first time, either by calling a getter method or by navigating the object graph.

|  |  |
| --- | --- |
| @Entity  public class Employee {  @Id  private int id;  private String name;  @ManyToOne(fetch = FetchType.LAZY)  private Department department;  // ...} | @Entity  public class Department {  @Id  private int id;  private String name;  // ...} |

In this example, we have two entities: Employee and Department. The Employee entity has a many-to-one relationship with the Department entity. By using the @ManyToOne annotation and setting the fetch attribute to FetchType.LAZY, we are telling Hibernate to use lazy fetching for the Department association. This means that the Department data will only be loaded from the database when it is needed, for example when we call the getDepartment() method on an Employee object.

# HQL

HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with [persistent objects](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/) and their properties. This is the main difference between HQL vs SQL.

* HQL is a superset of the JPQL, the Java Persistence Query Language. A JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries.
* HQL is a language with its own syntax and grammar. It is written as strings, like “from Product p“.
* HQL queries are translated by Hibernate into conventional SQL queries. Note that hibernate also provides the APIs that allow us to directly issue SQL queries as well.

Please note that Hibernator’s query facilities do not allow us to alter the database schema. We can only add/update/delete the data inside tables.

## Query interface

The Query Interface in Hibernate is an API used to execute HQL and SQL queries, allowing for convenient and flexible retrieval of data from a database and mapping it to Java objects. It provides methods for setting query parameters, retrieving results, and controlling query execution, making it database independent and a valuable tool for working with data in Hibernate. The Query Interface also supports Criteria Queries for building dynamic and flexible queries. It is a key component of the Hibernate framework and provides a high-level, object-oriented way to query relational databases.

## HQL Syntax

HQL syntax is defined as an [ANTLR](https://en.wikipedia.org/wiki/ANTLR) grammar. The grammar files are included in the grammar directory of the Hibernate core download. (ANTLR is a tool for building language parsers).

## Let us outline the syntax for the four fundamental CRUD operations here:

### HQL UPDATE Statement

UPDATE alters the details of existing objects in the database. In-memory entities, managed or not, will not be updated to reflect changes resulting from issuing UPDATE statements.

UPDATE [VERSIONED]

[FROM] path [[AS] alias] [, ...]

SET property = value [, ...]

[WHERE logicalExpression]

* path – fully qualified name of the entity or entities
* alias – used to abbreviate references to specific entities or their properties, and must be used when property names in the query would otherwise be ambiguous.
* VERSIONED – means that the update will update timestamps, if any, that are part of the entity being updated.
* property – names of properties of entities listed in the FROM path.
* logicalExpression – a where clause.

An example of the update statement. In this example, we are updating Employee with HQL update query multiple columns.

Query query=session.createQuery("update Employee set age=:age where name=:name");

query.setInteger("age", 32);

query.setString("name", "Lokesh Gupta");

int modifications=query.executeUpdate();

### HQL DELETE Statement

DELETE removes the details of existing objects from the database. In-memory entities will not be updated to reflect changes resulting from DELETE statements.

This also means that Hibernate’s cascade rules will not be followed for deletions carried out using HQL. However, if you have specified cascading deletes at the database level (either directly or through Hibernate, using the @OnDelete annotation), the database will still remove the child rows.

DELETE

[FROM] path [[AS] alias]

[WHERE logicalExpression]

In practice, deletes might look like this:

Query query=session.createQuery("delete from Account where acctStatus=:status");

query.setString("acctStatus", "PURGED");

int rowsDeleted=query.executeUpdate();

### HQL INSERT Statement

An HQL INSERT can be used to directly insert arbitrary entities as well as insert entities constructed from information obtained from SELECT queries.

INSERT-SELECT statements

INSERT INTO EntityName

properties\_list

SELECT select\_list

FROM ...

INSERT-VALUES statements

INSERT INTO EntityName

properties\_list

VALUES values\_list

The name of an entity is the path. The property names are the names of properties of entities listed in the FROM path of the incorporated SELECT query.

Given below is an example of copying users to a purged table before actually purging them :

Query query=session.createQuery("insert into purged\_accounts(id, code, status) "+

"Select id, code, status from account where accStatus=:status");

query.setString("accStatus", "PURGED");

int rowsCopied=query.executeUpdate();

### HQL SELECT Statement

An HQL SELECT is used to query the database for classes and their properties.

[SELECT [DISTINCT] property [, ...]]

FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]

WHERE logicalExpression

GROUP BY property [, ...]

HAVING logicalExpression

ORDER BY property [ASC | DESC] [, ...]

* The fully qualified name of an entity is path. The alias names may be used to abbreviate references to specific entities or their properties and must be used when property names used in the query would otherwise be ambiguous.
* The property names are the names of properties of entities listed in the FROM path.
* If FETCH ALL PROPERTIES is used then lazy loading semantics will be ignored, and all the immediate properties of the retrieved object(s) will be actively loaded (this does not apply recursively).
* WHERE is used to create HQL select query with WHERE clause.

When the properties listed consist only of the names of aliases in the FROM clause, the SELECT clause can be omitted in HQL. If we are using the JPA with JPQL, one of the differences between HQL and JPQL is that the SELECT clause is required in JPQL.

Given below are a few examples:

// 1

from Account FETCH ALL PROPERTIES

// 2

Select a.id, a.name, a.status from Account an order by a.id

// 3

Select a.id, a.name, a.status from Account a where a.id > 100

## FROM Clause and Aliases

The most important feature in HQL to note is the alias. Hibernate allows us to assign aliases to the classes in our query with the as a clause. Use the aliases to refer back to the class inside the query.

from Product as p

//or

from Product as product

The 'as' keyword is optional. We can also specify the alias directly after the class name as follows:

from Product product

If we need to fully qualify a class name in HQL, just specify the package and class name. Hibernate will take care of most of this behind the scenes, so we really need this only if we have classes with duplicate names in our application. If we have to do this in Hibernate, use syntax such as the following:

from com.howtodoinjava.geo.usa.Product

The from clause is very basic and useful for working directly with objects. However, if you want to work with the object’s properties without loading the full objects into memory, you must use the SELECT clause as explained in the next section.

## HQL SELECT Clause and Projections

The SELECT clause provides more control over the result set than the from clause. If you want to obtain the properties of objects in the result set, use the SELECT clause.

For instance, we could run a projection query on the products in the database that only returned the names, instead of loading the full object into memory, as follows:

select product.name from Product product

The result set for this query will contain a List of java.lang.String objects. Additionally, we can retrieve the prices and the names for each product in the database, like so:

select product.name, product.price from Product product

If you’re only interested in a few properties, this approach can allow you to reduce network traffic to the database server and save memory on the application’s machine.

## Named Parameters

Hibernate supports named parameters in its HQL queries. This makes writing queries that accept input from the user easy—and you do not have to defend against SQL injection attacks.

When using JDBC query parameters, any time you add, change or delete parts of the SQL statement, you need to update your Java code that sets its parameters because the parameters are indexed based on the order in which they appear in the statement.

Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters around in the query.

The simplest example of named parameters uses regular SQL types for the parameters:

String hql = "from Product where price > :price";

Query query = session.createQuery(hql);

query.setDouble("price",25.0);

List results = query.list();

## Paging through ResultSet

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user.

The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the Query interface for paging: setFirstResult() and setMaxResults(). The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the setMaxResults() method.

Your HQL will be unchanged while applying the caching — you need to only modify the Java code that executes the query.

Query query = session.createQuery("from Product");

query.setFirstResult(1);

query.setMaxResults(2);

List results = query.list();

displayProductsList(results);

If you turn on SQL logging, you can see which SQL commands Hibernate uses for pagination. For the open-source HSQLDB database, hibernate uses TOP and LIMIT.

Microsoft SQL Server does not support the LIMIT command, so hibernate uses only the TOP command. If your application is having performance problems with pagination, this can be very helpful for debugging.

If you only have one result in your HQL result set, hibernate has a shortcut method for obtaining just that object as discussed next.

## Getting a Unique Single Result

HQL’s Query interface provides a uniqueResult() method for obtaining just one object from an HQL query. Although your query may yield only one object, you may also use the uniqueResult() method with other result sets if you limit the results to just the first result. You could use the setMaxResults() method discussed in the previous section.

The uniqueResult() method on the Query object returns a single object, or null if there are zero results. If there is more than one result, then the uniqueResult() method throws a NonUniqueResultException.

String hql = "from Product where price > 25.0";

Query query = session.createQuery(hql);

query.setMaxResults(1);

Product product = (Product) query.uniqueResult();

**Sorting the Result**

To sort your HQL query’s 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: either ascending (asc) or descending (desc). You can use order on more than one property in the query if you need to.

from Product p where p.price&gt;25.0 order by p.price desc

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas.

For instance, you could sort by product price and the supplier’s name, as follows:

from Product p order by p.supplier.name asc, p.price asc

**HQL Associations**

Associations allow you to use more than one class in an HQL query, just as SQL allows you to use joins between tables in a relational database.

Hibernate supports five different types of joins:

* INNER JOIN
* CROSS JOIN
* LEFT OUTER JOIN
* RIGHT OUTER JOIN
* FULL OUTER JOIN

If you use cross join, just specify both classes in the from clause (from Product p, Supplier s). For the other joins, use a join clause after the from clause. Specify the type of join, the object property to join on, and an alias for the other class.

You can use an inner join to obtain the supplier for each product, and then retrieve the supplier's name, product name, and product price, as so:

select s.name, p.name, p.price from Product p inner join p.supplier as s

You can retrieve the objects using the similar syntax:

|  |
| --- |
| from Product p inner join p.supplier as s |

**HQL Aggregate Methods**

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects.

You may use the count(\*) syntax to count all the objects in the result set, or count(product.name) to count the number of objects in the result set with a name property. Here is an example using the count(\*) method to count all products:

select count(\*) from Product product

The aggregate functions available through HQL include the following:

1. avg(property name): The average of a property’s value
2. count(property name or \*): The number of times a property occurs in the results
3. max(property name): The maximum value of the property values
4. min(property name): The minimum value of the property values
5. sum(property name): The sum total of the property values

**Named Queries**

Named queries are created via class-level annotations on entities; normally, the queries apply to the entity in whose source file they occur, but there’s no absolute requirement for this to be true.

Named queries are created with the @NamedQueries annotation, which contains an array of @NamedQuery sets; each has a query and a name.

An example of named queries may look like this:

@NamedQueries({

@NamedQuery(name = "supplier.findAll", query = "from Supplier s"),

@NamedQuery(name = "supplier.findByName",

query = "from Supplier s where s.name=:name"),

})

Executing above named query is even simpler.

Query query = session.getNamedQuery("supplier.findAll");

List<Supplier> suppliers = query.list();

Read More – [Hibernate named query tutorial](https://howtodoinjava.com/hibernate/hibernate-named-query-tutorial/)

**Native SQL Queries**

Although you should probably use HQL whenever possible, hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL. Another reason is that you may want to call stored procedures from your Hibernate application.

You can modify your SQL statements to make them work with Hibernate’s ORM layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with {objectname.\*}, or you can specify the aliases directly with {objectname.property}.

Hibernate uses the mappings to translate your object property names into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support.

You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables so that you select the right properties off the table.

Underlying Hibernate’s native SQL support is the org.hibernate.SQLQuery interface, which extends the org.hibernate.Query interface. Your application will create a native SQL query from the session with the createSQLQuery() method on the Session interface.

public SQLQuery createSQLQuery(String queryString) throws HibernateException

After you pass a string containing the SQL query to the createSQLQuery() method, you should associate the SQL result with an existing Hibernate entity, a join, or a scalar result. The SQLQuery interface has addEntity(), addJoin(), and addScalar() methods.

Using native SQL with scalar results is the simplest way to get started with native SQL. Sample Java code looks like this:

String sql = "select avg(product.price) as avgPrice from Product product";

SQLQuery query = session.createSQLQuery(sql);

query.addScalar("avgPrice",Hibernate.DOUBLE);

List results = query.list();

A bit more complicated than the previous example is the native SQL that returns a result set of objects. In this case, we will need to map an entity to the SQL query.

String sql = "select {supplier.\*} from Supplier supplier";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity("supplier", Supplier.class);

List results = query.list();

//Hibernate modifies the SQL and executes the following command against the database:

select Supplier.id as id0\_, Supplier.name as name2\_0\_ from Supplier supplie

**Enabling Logs and Comments**

Hibernate can output the underlying SQL behind your HQL queries into your application’s log file. This is especially useful if the HQL query does not give the results you expect, or if the query takes longer than you wanted. This is not a feature you will have to use frequently, but it is useful should you have to turn to your database administrators for help in tuning your Hibernate application.

**HQL Logs**

The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the “show\_sql” property. Set this property to true in your hibernate.cfg.xml configuration file and hibernate will output the SQL into the logs. When you look in your application’s output for the Hibernate SQL statements, they will be prefixed with “Hibernate:”.

If you turn your log4j logging up to debug for the Hibernate classes, you will see SQL statements in your log files, along with lots of information about how Hibernate parsed your HQL query and translated it into SQL.

**HQL Comments**

Tracing your HQL statements through to the generated SQL can be difficult, so hibernate provides a commenting facility on the Query object that lets you apply a comment to a specific query. The Query interface has a setComment() method that takes a String object as an argument, as follows:

public Query setComment(String comment)

Hibernate will not add comments to your SQL statements without some additional configuration, even if you use the setComment() method. You will also need to set a Hibernate property, hibernate.use\_sql\_comments, to true in your Hibernate configuration.

If you set this property but do not set a comment on the query programmatically, hibernate will include the HQL used to generate the SQL call in the comment. I find this to be very useful for debugging HQL.

Use commenting to identify the SQL output in your application’s logs if SQL logging is enabled.

# Hibernate cascading

Cascading in Hibernate refers to the propagation of an operation from a parent entity to its child entities. For example, when you persist a parent entity, the associated child entities are automatically persisted as well, which is known as cascading persist.

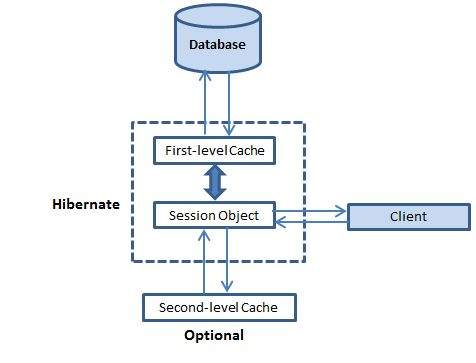
Hibernate supports various cascading options, such as "all", "save-update", "delete", "delete-orphan", "merge", etc. The choice of cascading option depends on the use-case and the desired behaviour.

Here are some common cascading options:

1. **all**: includes all cascading options
2. **save**-update: cascades changes to a child entity when the parent entity is updated
3. **delete**: cascades deletion of a parent entity to its child entities
4. **delete-orphan**: deletes a child entity when it's removed from its parent entity
5. **merge**: cascades the merge operation from a parent entity to its child entities.

By default, hibernate does not cascade any operations. To enable cascading, it has to be explicitly specified on the parent entity's mapping

# Hibernate Caching

Hibernate caching refers to the use of caching mechanisms to improve the performance of Hibernate-based applications by reducing the number of database hits. Hibernate provides two levels of caching:

## First-Level Cache

Also known as the Session cache, it is associated with a Hibernate Session and is enabled by default. It is responsible for caching objects within a single Hibernate session. When an object is retrieved from the database, hibernate stores it in the first-level cache, and subsequent requests for the same object are served from the cache instead of hitting the database.

Example:

Session session = sessionFactory.openSession();  
Transaction tx = session.beginTransaction();  
// Load a customer object from the database  
Customer customer = session.get(Customer.class, 1L);  
// This will not hit the database since customer object is already in the first-level cache  
customer = session.get(Customer.class, 1L);  
tx.commit();  
session.close();

## Second-Level Cache

Also known as the Session Factory cache, it is shared by all sessions created by a Session Factory and is optional. It can be used to cache data across multiple sessions and improve performance further. Hibernate supports several caching providers, including Ehcache, Infinispan, and Hazelcast.

Example:

// Configure Hibernate to use second-level cache  
<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property> <property name="hibernate.cache.use\_second\_level\_cache">true</property>  
// Annotate a persistent class to use second-level cache  
@Cacheable  
@Entity  
public class Customer {  
 ...  
}

In conclusion, hibernate caching can significantly improve the performance of Hibernate-based applications by reducing the number of database hits. However, caching should be used judiciously, as it may introduce additional complexity and overhead.

Caching is a mechanism to enhance the performance of a system. It is a buffer memorythat lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Caching is important to Hibernate as well. It utilizes a multilevel caching scheme as explained below −

## Query-level Cache

Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

## The Second Level Cache

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache.

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.

## Concurrency Strategies

A concurrency strategy is a mediator, which is responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

Transactional − Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

Read-write − Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

Nonstrict-read-write − This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.

Read-only − A concurrency strategy suitable for data, which never changes. Use it for reference data only.

If we are going to use second-level caching for our Employee class, let us add the mapping element required to tell Hibernate to cache Employee instances using read-write strategy.

<?xml version = "1.0" encoding = "utf-8"?> <!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"> <hibernate-mapping> <class **name** = "Employee" **table** = "EMPLOYEE"> <meta **attribute** = "class-description"> This class contains the employee detail.   
 </meta> <cache **usage** = "read-write"/> <id **name** = "id" **type** = "int" **column** = "id"> <generator **class**="native"/> </id> <property **name** = "firstName" **column** = "first\_name" **type** = "string"/> <property **name** = "lastName" **column** = "last\_name" **type** = "string"/> <property **name** = "salary" **column** = "salary" **type** = "int"/> </class> </hibernate-mapping>

The usage="read-write" attribute tells Hibernate to use a read-write concurrency strategy for the defined cache.

Cache Provider

Your next step after considering the concurrency strategies, you will use your cache candidate classes to pick a cache provider. Hibernate forces you to choose a single cache provider for the whole application.

|  |  |
| --- | --- |
| Sr.No. | Cache Name & Description |
| 1 | EHCache  It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache. |
| 2 | OSCache  Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support. |
| 3 | warmCache  A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache. |
| 4 | JBoss Cache  A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking. The Hibernate query cache is supported. |

Every cache provider is not compatible with every concurrency strategy. The following compatibility matrix will help you choose an appropriate combination.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strategy/Provider | Read-only | Nonstrictread-write | Read-write | Transactional |
| EHCache | X | X | X |  |
| OSCache | X | X | X |  |
| SwarmCache | X | X |  |  |
| JBoss Cache | X |  |  | X |

You will specify a cache provider in hibernate.cfg.xml configuration file. We choose EHCache as our second-level cache provider −

<property name = "hibernate.connection.password">  
 root123  
 </property>  
 <property name = "hibernate.cache.provider\_class">  
 org.hibernate.cache.EhCacheProvider  
 </property>  
 <!-- List of XML mapping files -->  
 <mapping resource = "Employee.hbm.xml"/>  
 </session-factory>  
</hibernate-configuration>

Now, you need to specify the properties of the cache regions. EHCache has its own configuration file, **ehcache.xml**, which should be in the CLASSPATH of the application. A cache configuration in ehcache.xml for the Employee class may look like this −

<diskStore path="java.io.tmpdir"/> <defaultCache maxElementsInMemory = "1000" eternal = "false" timeToIdleSeconds = "120" timeToLiveSeconds = "120" overflowToDisk = "true" /> <cache name = "Employee" maxElementsInMemory = "500" eternal = "true" timeToIdleSeconds = "0" timeToLiveSeconds = "0" overflowToDisk = "false" />

That's it, now we have second-level caching enabled for the Employee class and Hibernate, now hits the second-level cache whenever you navigate to an Employee or when you load an Employee by identifier.

You should analyze your all the classes and choose appropriate caching strategy for each of the classes. Sometime, second-level caching may downgrade the performance of the application. So, it is recommended to benchmark your application first, without enabling caching and later on enable your well suited caching and check the performance. If caching is not improving system performance, then there is no point in enabling any type of caching.

## The Query-level Cache

To use the query cache, you must first activate it using the **hibernate.cache.use\_query\_cache="true"** property in the configuration file. By setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Next, to use the query cache, you use the setCacheable(Boolean) method of the Query class. For example −

Session session = SessionFactory.openSession();  
Query query = session.createQuery("FROM EMPLOYEE");  
query.setCacheable(true);  
List users = query.list();  
SessionFactory.closeSession();

Hibernate also supports very fine-grained cache support through the concept of a cache region. A cache region is part of the cache that's given a name.

Session session = SessionFactory.openSession();  
Query query = session.createQuery("FROM EMPLOYEE");  
query.setCacheable(true);  
query.setCacheRegion("employee");  
List users = query.list();  
SessionFactory.closeSession();

This code uses the method to tell Hibernate to store and look for the query in the employee area of the cache.

# Named Queries

A Named Query in Hibernate is a predefined query that is associated with a name and stored in the mapping metadata. Named Queries allow you to define a reusable, named query in a centralized location, typically in a mapping file or via annotations, rather than hard-coding the HQL (Hibernate Query Language) or SQL in your code. This makes your code more maintainable, as you can change the underlying query without affecting the application code that uses it. Named Queries can be defined using either HQL or native SQL and can be executed using the getNamedQuery() method of the Session object in Hibernate. Named Queries can be used to retrieve data from the database or to perform data modification operations such as updates and deletes.

### Advantages

The named queries help in grouping the HQL statements in a single place and lately refer to them using pre-configured names whenever we need to use them. This grouping helps largely in code cleanup because these HQL statements are no longer scattered in the whole code.

#### Apart from the above, there are some more advantages of named queries:

1. **Fail fast:** query syntaxes are checked when the [SessionFactory](https://howtodoinjava.com/hibernate/hibarnate-build-sessionfactory/) is created, making the application fail fast in case of an error.
2. **Reusable:** These can be accessed and used from several places in the application which increases re-usability.

#### Performance-wise named queries do not make much difference, nor put any excessive cost.

1. The *cost of transforming an* HQL query to SQL is negligible compared to the cost of actually executing the query.
2. The *memory cost of caching* the queries is really small. Remember that Hibernate needs to have all the entities’ meta-data in memory anyway.

## Creating @NamedQuery

To demonstrate how to declare a named query, we have the *DepartmentEntity*. We are creating a simple named query that fetches a department by its id.

Named query definition has two important attributes:

* **name**: The name of the named query by which it will be located using the *Session* or *EntityManager* interface.
* **query**: The HQL or SQL statement to get executed in the database.

@NamedQuery(name="findCustomerByName",

query="FROM Customer c WHERE c.name = :name")

@Entity

public class Customer {

// Fields, constructors, getters, and setters

}

The @NamedQuery annotation is used to define a named query, which can be re-used multiple times throughout the application. In this example, the named query is called "findCustomerByName" and it retrieves a Customer entity based on the name.

The query can be executed using the following code:

EntityManager entityManager = ...;

TypedQuery<Customer> query = entityManager.createNamedQuery("findCustomerByName", Customer.class);

query.setParameter("name", "John Doe");

List<Customer> customers = query.getResultList();

## Grouping with *@NamedQueries*

If we have multiple named queries for an entity we can group them using the *@NamedQueries* annotation.

@NamedQueries({

@NamedQuery(name = "QUERY\_GET\_DEPARTMENT\_BY\_ID",

query = "from DepartmentEntity d where d.id = :id"),

@NamedQuery(name = "QUERY\_UPDATE\_DEPARTMENT\_BY\_ID",

query = "UPDATE DepartmentEntity d SET d.name=:name where d.id = :id")

})

@Entity

public class DepartmentEntity implements Serializable {

public static final String QUERY\_GET\_DEPARTMENT\_BY\_ID

= "QUERY\_GET\_DEPARTMENT\_BY\_ID";

public static final String QUERY\_UPDATE\_DEPARTMENT\_BY\_ID

= "QUERY\_UPDATE\_DEPARTMENT\_BY\_ID";

//...

}

## Executing a Named Query

To execute a named query, we can use the *createNamedQuery()* method that creates a *Query* instance. Then we can use a method from *Query* interface to execute the SQL query.

@Test

public void getEntityById() {

Query query =

em.createNamedQuery(DepartmentEntity.QUERY\_GET\_DEPARTMENT\_BY\_ID)

.setParameter("id", 1);

DepartmentEntity dept = (DepartmentEntity) query.getSingleResult();

Assertions.assertEquals("HR", dept.getName());

}

## Using @NamedNativeQuery

The *@NamedNativeQuery* works very similar to *@NamedQuery* except we need to write the native SQL statements instead of HQL.

@NamedNativeQueries({

@NamedNativeQuery(name = "NATIVE\_QUERY\_GET\_DEPARTMENT\_BY\_ID",

query = "SELECT \* FROM TBL\_DEPT d WHERE d.id = :id"),

@NamedNativeQuery(name = "NATIVE\_QUERY\_UPDATE\_DEPARTMENT\_BY\_ID",

query = "UPDATE TBL\_DEPT d SET d.name=:name WHERE d.id = :id")

})

The execution of named native queries is pretty much similar to the named queries.

@Test

public void updateEntityByIdWithNamedNativeQuery() {

Query query =

em.createNamedQuery(DepartmentEntity.NATIVE\_QUERY\_UPDATE\_DEPARTMENT\_BY\_ID)

.setParameter("name", "SUPPORT")

.setParameter("id", 1);

query.executeUpdate();

flushAndClear();

DepartmentEntity dept = em.find(DepartmentEntity.class, 1);

Assertions.assertEquals("SUPPORT", dept.getName());

}

## Conclusion

In this tutorial, we learned to create, group and execute named queries using hibernate. As a best practice we should:

* Use native queries preferably only for selecting records based on complex conditions. Do not use them excessively, otherwise, there is no use of using ORM over simple JDBC.
* Remember that result of named queries is not cached in the secondary cache, only the query object itself gets cached.
* Make a habit of adding a couple of unit testcases whenever we add any named query in code. And run those unit testcases immediately.
* Fortunately, we can not have two named queries with the same name in hibernate. Hibernate shows fail-fast behavior in this regard and will show errors in the application start-up itself.

# JPA Criteria API

* JPA Criteria Queries is a feature in Java Persistence API (JPA) for creating and executing database queries in a type-safe, dynamic, and object-oriented way.
* The JPA Criteria API provides a programmatic approach for building queries, instead of using string-based JPQL or SQL.
* Queries can be built by defining criteria for entity selection, conditions, ordering, and grouping.
* The Criteria API uses metadata to validate the criteria at compile time, making it less prone to errors compared to string-based queries.
* Hibernate also provides Criteria API as a method of manipulating objects and data in RDBMS tables.
* The Hibernate Session interface provides a createCriteria() method for creating a Criteria object.
* The Criteria object returns instances of the persistence object's class when the application executes the criteria query.

## Simple criteria example

Session session = HibernateConfig.getSessionFactory().openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> root = criteriaQuery.from(Employee.class);

criteriaQuery.select(root);

criteriaQuery.where(criteriaBuilder.equal(root.get("eName"), "Schiller,Toney"));

TypedQuery<Employee> query = session.createQuery(criteriaQuery);

List<Employee> employees = query.getResultList();

for(Employee e : employees)

System.out.println(e);

## JPA Criteria notes

### CriteriaBuilder

The starting point for building a criteria query is the CriteriaBuilder interface. This interface provides a set of methods for constructing various types of criteria expressions, such as predicates, expressions, and ordering. The CriteriaBuilder can be obtained from the EntityManager.

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

### CriteriaQuery

The CriteriaQuery interface is used to define the criteria query. It allows you to specify the entity you are querying, what fields to select, and what conditions to apply to the query.

CriteriaQuery<Employee> criteriaQuery = cb.createQuery(Employee.class);

### Root

The Root interface represents the root of the criteria query. It provides access to the entity you are querying and its properties.

Root<Employee> employee = criteriaQuery.from(Employee.class);

### Selecting Fields

To specify what fields to select in the query, use the select() method of the CriteriaQuery.

criteriaQuery.select(employee);

### Conditions

To add conditions to the query, you can use the where() method of the CriteriaQuery. The where() method takes a Predicate as an argument, which represents a Boolean expression. Predicates can be created using the methods of the CriteriaBuilder.

criteriaQuery.where(cb.equal(employee.get("age"), 30));

### Joining tables

To join tables, you can use the join() method of the Root.

Join<Employee, Department> department = employee.join("department");

### Ordering

To order the results of the query, you can use the orderBy() method of the CriteriaQuery.

criteriaQuery.orderBy(cb.asc(employee.get("age")));

### Executing the Query

Once you have built the CriteriaQuery, you can execute it by creating a TypedQuery from the EntityManager and executing the getResultList() method.

TypedQuery<Employee> query = entityManager.createQuery(criteriaQuery);

List<Employee> employees = query.getResultList();

These are the basics of using JPA Criteria API in Java. With its powerful and flexible API, you can build complex, type-safe queries with ease.

1. Obtain a **CriteriaBuilder** instance: The **CriteriaBuilder** is the main entry point for building criteria queries. It can be obtained from the **EntityManager** by calling **entityManager.getCriteriaBuilder()**.
2. Create a **CriteriaQuery** instance: A **CriteriaQuery** represents the query itself. It can be created by calling **criteriaBuilder.createQuery(ResultType.class)**, where **ResultType** is the expected type of the query result.
3. Define the **Root** instance: A **Root** represents the entry point for all the attributes of the entity being queried. It can be created by calling **criteriaQuery.from(EntityClass.class)**, where **EntityClass** is the entity class being queried.
4. Specify the query selection: The query selection defines what attributes or expressions will be included in the query result. It can be specified by calling **criteriaQuery.select(root)**, where **root** is the **Root** instance.
5. Define the query filter: The query filter defines the conditions that must be met for an entity to be included in the query result. It can be defined by creating a **Predicate** and adding it to the query using **criteriaQuery.where(predicate)**. The **Predicate** can be created by calling various methods on the **CriteriaBuilder**, such as **greaterThan**, **equal**, etc.
6. Create a **TypedQuery** instance: A **TypedQuery** is a query that has a specified result type. It can be created by calling **entityManager.createQuery(criteriaQuery)**.
7. Execute the query: The query can be executed by calling the **getResultList** or **getSingleResult** method on the **TypedQuery** instance.

Here's an example that demonstrates these steps:

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = cb.createQuery(Employee.class); Root<Employee> employeeRoot = criteriaQuery.from(Employee.class);

criteriaQuery.select(employeeRoot);

Predicate salaryPredicate = cb.greaterThan(employeeRoot.get("salary"), 10000);

criteriaQuery.where(salaryPredicate); TypedQuery<Employee> query = entityManager.createQuery(criteriaQuery); List<Employee> resultList = query.getResultList();

In this example, we obtain a **CriteriaBuilder** instance, create a **CriteriaQuery** for the **Employee** class, define the **Root** instance for the **Employee** class, specify the query selection, create a **Predicate** to filter employees with a salary greater than 10000, create a **TypedQuery**, and finally execute the query by calling the **getResultList** method on the **TypedQuery**. The result is a list of **Employee** objects that satisfy the specified criteria.

**Ex2-**

Session session = HibernateConfig.getSessionFactory().openSession();  
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);  
Root root = criteriaQuery.from(Employee.class);  
criteriaQuery.select(root);  
TypedQuery<Employee> query=session.createQuery(criteriaQuery);  
List<Employee> employees = query.getResultList();  
for(Employee e : employees)  
 System.out.println(e);

* The **Session** object is obtained from the **HibernateConfig.getSessionFactory().openSession()** method. The Session is a primary interface for the Hibernate framework and it provides methods for performing CRUD operations on the database.
* The **CriteriaBuilder** object is obtained from the Session using the **session.getCriteriaBuilder()** method. The **CriteriaBuilder** is used to programmatically build a **CriteriaQuery** in a type-safe manner.
* The **CriteriaQuery** is created using the **criteriaBuilder.createQuery(Employee.class)** method. It defines the type of objects that will be returned by the query.
* The Root object is created using the **criteriaQuery.from(Employee.class)** method. The Root represents the root of the entity and it is used to specify what to select from the entity.
* The **criteriaQuery**.select(root) method is used to specify what to select from the entity. In this case, the entire Employee entity is selected.
* Finally, the **TypedQuery** is created using the **session.createQuery(criteriaQuery)** method. The **TypedQuery** is used to execute the **CriteriaQuery** and retrieve the result as a list of Employee objects.

# Overview

It enables us to write queries without doing raw SQL as well as gives us some object-oriented control over the queries, which is one of the main features of Hibernate. The Criteria API allows us to build up a criteria query object programmatically, where we can apply different kinds of filtration rules and logical conditions.

Since Hibernate 5.2, the Hibernate Criteria API is deprecated, and new development is focused on the JPA Criteria API. We'll explore how to use Hibernate and JPA to build Criteria Queries.

2. Maven Dependencies

To illustrate the API, we'll use the reference JPA implementation Hibernate.

To use Hibernate, we'll make sure to add the latest version of it to our pom.xml file:

<dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-core</artifactId>   
 <version>5.3.2.Final</version>  
</dependency>

## 3. Simple Example Using Criteria

Let's start by looking at how to retrieve data using Criteria Queries. We'll look at how to get all the instances of a particular class from the database.

We have an Item class that represents the tuple “ITEM” in the database:

public class Item implements Serializable {  
 private Integer itemId;  
 private String itemName;  
 private String itemDescription;  
 private Integer itemPrice;  
 // standard setters and getters  
}

Let's look at a simple criteria query that will retrieve all the rows of “ITEM” from the database:

Session session = HibernateUtil.getHibernateSession();  
CriteriaBuilder cb = session.getCriteriaBuilder();  
CriteriaQuery<Item> cr = cb.createQuery(Item.class);  
Root<Item> root = cr.from(Item.class);  
cr.select(root);  
  
Query<Item> query = session.createQuery(cr);  
List<Item> results = query.getResultList();Copy

The above query is a simple demonstration of how to get all the items. Let's see it step by step:

Create an instance of Session from the SessionFactory object

Create an instance of CriteriaBuilder by calling the getCriteriaBuilder() method

Create an instance of CriteriaQuery by calling the CriteriaBuilder createQuery() method

Create an instance of Query by calling the Session createQuery() method

Call the getResultList() method of the query object, which gives us the results

Now that we've covered the basics, let's move on to some of the features of criteria query.

## 3.1. Using Expressions

The CriteriaBuilder can be used to restrict query results based on specific conditions, by using CriteriaQuery where() method and providing Expressions created by CriteriaBuilder.

Let's see some examples of commonly used Expressions.

In order to get items having a price of more than 1000:

cr.select(root).where(cb.gt(root.get("itemPrice"), 1000));

Next, getting items having itemPrice less than 1000:

cr.select(root).where(cb.lt(root.get("itemPrice"), 1000));

Items having itemName contain Chair:

cr.select(root).where(cb.like(root.get("itemName"), "%chair%"));

Records having itemPrice between 100 and 200:

cr.select(root).where(cb.between(root.get("itemPrice"), 100, 200));

Items having itemName in Skate Board, Paint and Glue:

cr.select(root).where(root.get("itemName").in("Skate Board", "Paint", "Glue"));

To check if the given property is null:

cr.select(root).where(cb.isNull(root.get("itemDescription")));

To check if the given property is not null:

cr.select(root).where(cb.isNotNull(root.get("itemDescription")));

We can also use the methods isEmpty() and isNotEmpty() to test if a List within a class is empty or not.

Additionally, we can combine two or more of the above comparisons. The Criteria API allows us to easily chain expressions:

Predicate[] predicates = new Predicate[2];  
predicates[0] = cb.isNull(root.get("itemDescription"));  
predicates[1] = cb.like(root.get("itemName"), "chair%");  
cr.select(root).where(predicates);

To add two expressions with logical operations:

Predicate greaterThanPrice = cb.gt(root.get("itemPrice"), 1000);  
Predicate chairItems = cb.like(root.get("itemName"), "Chair%");

Items with the above-defined conditions joined with Logical OR:

cr.select(root).where(cb.or(greaterThanPrice, chairItems));

To get items matching with the above-defined conditions joined with Logical AND:

cr.select(root).where(cb.and(greaterThanPrice, chairItems));

## 3.2. Sorting

Now that we know the basic usage of Criteria, let's look at the sorting functionalities of Criteria.

In the following example, we order the list in ascending order of the name and then in descending order of the price:

cr.orderBy(  
 cb.asc(root.get("itemName")), cb.desc(root.get("itemPrice")));

In the next section, we will have a look at how to do aggregate functions.

## 3.3. Projections, Aggregates and Grouping Functions

Now let's see the different aggregate functions.

Get row count:

CriteriaQuery<Long> cr = cb.createQuery(Long.class);  
Root<Item> root = cr.from(Item.class);  
cr.select(cb.count(root));  
Query<Long> query = session.createQuery(cr);  
List<Long> itemProjected = query.getResultList();

The following is an example of aggregate functions — Aggregate function for Average:

CriteriaQuery<Double> cr = cb.createQuery(Double.class);  
Root<Item> root = cr.from(Item.class);  
cr.select(cb.avg(root.get("itemPrice")));  
Query<Double> query = session.createQuery(cr);  
List avgItemPriceList = query.getResultList();

Other useful aggregate methods are sum(), max(), min(), count(), etc.

## 3.4. CriteriaUpdate

Starting from JPA 2.1, there's support for performing database updates using the Criteria API.

CriteriaUpdate has a set() method that can be used to provide new values for database records:

CriteriaUpdate<Item> criteriaUpdate = cb.createCriteriaUpdate(Item.class);  
Root<Item> root = criteriaUpdate.from(Item.class);  
criteriaUpdate.set("itemPrice", newPrice);  
criteriaUpdate.where(cb.equal(root.get("itemPrice"), oldPrice));  
  
Transaction transaction = session.beginTransaction();  
session.createQuery(criteriaUpdate).executeUpdate();  
transaction.commit();

In the above snippet, we create an instance of CriteriaUpdate<Item> from the CriteriaBuilder and use its set() method to provide new values for the itemPrice. In order to update multiple properties, we just need to call the set() method multiple times.

## 3.5. CriteriaDelete

CriteriaDelete enables a delete operation using the Criteria API.

We just need to create an instance of CriteriaDelete and use the where() method to apply restrictions:

CriteriaDelete<Item> criteriaDelete = cb.createCriteriaDelete(Item.class);  
Root<Item> root = criteriaDelete.from(Item.class);  
criteriaDelete.where(cb.greaterThan(root.get("itemPrice"), targetPrice));  
  
Transaction transaction = session.beginTransaction();  
session.createQuery(criteriaDelete).executeUpdate();  
transaction.commit();

## 4. Advantage Over HQL

In the previous sections, we covered how to use Criteria Queries.

Clearly, the main and most hard-hitting advantage of Criteria Queries over HQL is the nice, clean, object-oriented API.

We can simply write more flexible, dynamic queries compared to plain HQL. The logic can be refactored with the IDE and has all the type-safety benefits of the Java language itself.

Of course, there are some disadvantages as well, especially around more complex joins.

So, we generally have to use the best tool for the job — that can be the Criteria API in most cases, but there are definitely cases where we'll have to go lower level.