**Hibernate**

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

ORM Tool

Object-Relational Mapping (ORM) is a technique that allows developers to map object-oriented code to a relational database system. It bridges the gap between the object-oriented programming paradigm used in application development and the relational database model used for data storage.



## What is JPA?

## JPA stands for Java Persistence API. It is a Java specification that defines a set of standards and interfaces for object-relational mapping (ORM) in Java applications. JPA provides a high-level, object-oriented approach to persisting data in relational databases.

## Advantages of Hibernate Framework

Following are the advantages of hibernate framework:

### **1) Open Source and Lightweight**

Hibernate framework is open source under the LGPL license and lightweight.

Open Source: Open source software refers to software that is released with a license that allows users to access, use, modify, and distribute the source code freely. The source code is made available to the public, fostering transparency, collaboration, and community-driven development. Open source projects often have vibrant communities, where developers contribute improvements, report issues, and share knowledge.

Lightweight: A lightweight software tool or framework is characterized by its small size, minimal resource requirements, and simplicity. Lightweight tools focus on providing essential features without excessive overhead or unnecessary complexity. They are designed to be efficient, easy to use, and performant.

### **2) Fast Performance**

The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

### **3) Database Independent Query**

HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

### **4) Automatic Table Creation**

Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

### **5) Simplifies Complex Join**

Fetching data from multiple tables is easy in hibernate framework.

# **Hibernate Architecture**

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

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





A persistent object, in the context of software development, refers to an object that can be stored and retrieved from a persistent storage medium, such as a database, in a way that allows it to retain its state across different program executions.

#### **SessionFactory**

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

In Hibernate, the **SessionFactory** is a key component that provides an interface between your application and the underlying database. It is responsible for creating and managing database connections, as well as mapping Java objects to database tables.

#### **Session**

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

#### **Transaction**

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

#### **ConnectionProvider**

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

#### **TransactionFactory**

It is a factory of Transaction. It is optional.

# **Caching in Hibernate**

Caching in Hibernate refers to the mechanism of storing frequently accessed data in memory to improve the performance of database operations. Hibernate provides several levels of caching that can be utilized to reduce the number of database queries and improve overall application performance.

There are mainly two types of caching:

* First Level Cache, and
* Second Level Cache

#### **First Level Cache**

The first level cache, also known as the session cache, is enabled by default in Hibernate.

* It operates at the session or EntityManager level, storing objects that have been queried or persisted within a particular session.

The EntityManager is a core component of the Java Persistence API (JPA) and is responsible for managing entities, persisting data to the database, retrieving data, and executing queries. It serves as an interface between the application and the persistence context, which represents a set of managed entities.

* The first level cache is managed by Hibernate and is transparent to the developer.
* When an entity is fetched or loaded within a session, it is stored in the first level cache. Subsequent requests for the same entity within the same session will be retrieved from the cache, eliminating the need for additional database queries.

#### **Second Level Cache**

* The second level cache is an optional level of caching that operates at the session factory or EntityManagerFactory level.

The EntityManagerFactory is a central component in the Java Persistence API (JPA) that represents a factory for creating EntityManager instances. It is responsible for initializing the JPA provider, establishing the connection to the database, and managing the persistence units.

* Objects that are frequently accessed or shared among different sessions can be stored in the second level cache, reducing the need for repeated database queries.
* The second level cache can be configured for specific entities or collections, and it provides caching strategies such as read-only, transactional, and non-strict read-write.

In Hibernate, both the first-level cache (also known as session-level cache) and second-level cache (also known as session factory-level cache) are used to improve performance by reducing the number of database queries. However, they differ in their scope, lifespan, and usage patterns. Here's an overview of the differences between the two:

First-Level Cache (Session-Level Cache):

Scope: The first-level cache is associated with the Hibernate Session object. It is local to the session and is not shared between different sessions.

Lifespan: The first-level cache is available as long as the session is open. When the session is closed, the first-level cache is cleared, and the objects are no longer accessible.

Purpose: The first-level cache is used to store the objects retrieved or persisted during a particular session. It acts as a transactional cache, ensuring consistency within a session.

Automatic: The first-level cache is maintained automatically by Hibernate. Whenever an entity is fetched or loaded, Hibernate checks the first-level cache first to see if the object is already present before issuing a database query.

Concurrency: The first-level cache is designed to work in a single-threaded or thread-safe environment, typically within a single session.

Second-Level Cache (Session Factory-Level Cache):

Scope: The second-level cache is shared among all sessions that are created from the same SessionFactory. It allows multiple sessions to share cached data.

Lifespan: The second-level cache persists as long as the SessionFactory remains active. It survives across multiple sessions and can be accessed by different users or processes.

Purpose: The second-level cache is used to cache frequently accessed or read-only data, such as reference data or static lookup tables. It helps reduce the number of database queries by storing the entities or query results in the cache.

Configuration: The second-level cache needs to be explicitly configured in Hibernate, including the cache provider (e.g., EHCache, Infinispan) and cache settings (e.g., cache regions, eviction policies).

Concurrency: The second-level cache is designed to handle concurrent access and supports multi-threaded and distributed environments.

In summary, the first-level cache is associated with a specific Hibernate session and provides transactional caching within that session. On the other hand, the second-level cache is shared among multiple sessions and is used for caching frequently accessed or read-only data across sessions. Both caches play important roles in improving performance and reducing database interactions, but their usage and behavior differ based on their scope and lifespan.

In software development, an entity represents a specific object . It is typically a persistent data structure that corresponds to a database. Entities encapsulate both data and behavior, allowing developers to work with objects that have properties and methods associated with them.

A Java class can be easily transformed into an entity. For transformation the basic requirements are: -

* No-argument Constructor
* Annotation

### @Data annotation

The **@Data annotation** is equivalent to the combination of the following annotations:

* @Getter
* @Setter
* @RequiredArgsConstructor
* @ToString
* @EqualsAndHashCode

We can replace annotating a class with the annotations that are listed above and a single @Data annotation. The @Data annotation does the following work:

* It generates the getter methods for all the fields.
* It generates the setter methods for all the non-final fields.
* It generates the toString() method implementation.
* It generates appropriate equals() and hashCode() implementations, involving the fields of class.
* It generates a constructor that initializes all the final fields, as well as all the non-final fields with no initializer that have been marked with @NonNull, in order to ensure that the field is never null.

Java equals()

* The java equals() is a method of *lang.Object* class, and it is used to compare two objects.
* To compare two objects that whether they are the same, it compares the values of both the object's attributes.
* By default, two objects will be the same only if stored in the same memory location.

Java hashcode()

* A **hashcode** is an integer value associated with every object in Java, facilitating the hashing in hash tables.
* The hashcode() method returns the same hash value when called on two objects, which are equal according to the equals() method. And if the objects are unequal, it usually returns different hash values.

The @GeneratedValue annotation in JPA is used to specify the strategy for generating unique identifier values for entities. When using the @GeneratedValue annotation with the GenerationType.SEQUENCE strategy, the primary key values are generated using a database sequence.

# **Generator classes in Hibernate**

All the generator classes implements the **org.hibernate.id.IdentifierGenerator**[**interface**](https://www.javatpoint.com/interface-in-java). The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface. Hibernate framework provides many built-in generator classes:

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

### **1) assigned**

It is the default generator strategy if there is no <generator> element . In this case, application assigns the id. For example:

### **2) increment**

It generates the unique id only if no other process is inserting data into this table. It generates **short**, **int** or **long** type identifier. If a table contains an identifier then the application considers its maximum value else the application consider that the first generated identifier is 1. For each attribute value, the hibernate increment the identifier by 1. Syntax

### **3) sequence**

It uses the sequence of the database. if there is no sequence defined, it creates a sequence automatically e.g. in case of Oracle database, it creates a sequence named HIBERNATE\_SEQUENCE.

### **4) hilo**

It uses high and low algorithm to generate the id of type short, int and long. Syntax:

### **5) native**

It uses identity, sequence or hilo depending on the database vendor. Syntax:

### **6) identity**

It is used in Sybase, My SQL, MS SQL Server, DB2 and HypersonicSQL to support the id column. The returned id is of type short, int or long. It is responsibility of database to generate unique identifier.

### **7) seqhilo**

It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.

### **8) uuid**

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

### **9) guid**

|  |
| --- |
| It uses GUID generated by database of type string. It works on MS SQL Server and MySQL. |

### **10) select**

|  |
| --- |
| It uses the primary key returned by the database trigger. |

### **11) foreign**

|  |
| --- |
| It uses the id of another associated object, mostly used with <one-to-one> association. |

### **12) sequence-identity**

|  |
| --- |
| It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only. |

# **SQL Dialects in Hibernate**

The dialect specifies the type of database used in hibernate so that hibernate generate appropriate type of SQL statements. For connecting any hibernate application with the database, it is required to provide the configuration of SQL dialect.

**Cascading**

Cascading is a feature in Hibernate, which is refers to the ability to propagate(automatically reflected) changes from one entity to another automatically. When an operation (such as save, update, or delete) is performed on a particular entity, cascading allows Hibernate to automatically apply the same operation to related entities.

Different Cascade Types in Hibernate

Hibernate provides several types of cascade options that can be used to manage the relationships between entities. Here are the different cascade types in Hibernate:

* CascadeType.ALL
* CascadeType.PERSIST
* CascadeType.MERGE
* CascadeType.REMOVE
* CascadeType.REFRESH
* CascadeType.DETACH
* CascadeType.REPLICATE
* CascadeType.SAVE\_UPDATE

1. CascadeType.ALL

* CascadeType.ALL is a cascading type in Hibernate that specifies that all state transitions (create, update, delete, and refresh) should be cascaded from the parent entity to the child entities.

@OneToMany(mappedBy="customer", cascade=CascadeType.ALL)

private Set<Order> orders;

* This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that all state transitions should be cascaded to the associated orders. This means that when a Customer entity is saved, updated, or deleted, all of its associated Order entities will also be saved, updated, or deleted accordingly.+

2. CascadeType.PERSIST

CascadeType.PERSIST is a cascading type in Hibernate that specifies that the create (or persist) operation should be cascaded from the parent entity to the child entities.

When CascadeType.PERSIST is used, any new child entities associated with a parent entity will be automatically persisted when the parent entity is persisted. However, updates or deletions made to the parent entity will not be cascaded to the child entities.

@OneToMany(mappedBy="customer", cascade=CascadeType.PERSIST)

private Set<Order> orders;

This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that the persistent operation should be cascaded to the associated orders. This means that when a Customer entity is persisted, any new Order entities associated with it will also be persisted automatically. However, updates or deletions made to the Customer entity will not be cascaded to the associated Order entities.

3. CascadeType.MERGE

CascadeType.MERGE is a cascading type in Hibernate that specifies that the update (or merge) operation should be cascaded from the parent entity to the child entities.

When CascadeType.MERGE is used, any changes made to a detached parent entity (i.e., an entity that is not currently managed by the persistence context) will be automatically merged with its associated child entities when the parent entity is merged back into the persistence context. However, new child entities that are not already associated with the parent entity will not be automatically persisted.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using CascadeType.MERGE, any changes made to a detached Customer entity (such as changes made in a different session or transaction) will be automatically merged with its associated Order entities when the Customer entity is merged back into the persistence context.

@OneToMany(mappedBy="customer", cascade=CascadeType.MERGE)

private Set<Order> orders;

This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that the merge operation should be cascaded to the associated orders. This means that when a detached Customer entity is merged back into the persistence context, any changes made to the Customer entity will be automatically merged with its associated Order entities. However, new Order entities that are not already associated with the Customer entity will not be automatically persisted.

4. CascadeType.REMOVE

CascadeType.REMOVE is a cascading type in Hibernate that specifies that the delete operation should be cascaded from the parent entity to the child entities.

When CascadeType.REMOVE is used, any child entities associated with a parent entity will be automatically deleted when the parent entity is deleted. However, updates or modifications made to the parent entity will not be cascaded to the child entities.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using CascadeType.REMOVE, any Order entities associated with a Customer entity will be automatically deleted when the Customer entity is deleted.

Here’s an example of how CascadeType.REMOVE can be used:

@OneToMany(mappedBy="customer", cascade=CascadeType.REMOVE)

private Set<Order> orders;

This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that the remove operation should be cascaded to the associated orders. This means that when a Customer entity is deleted, any associated Order entities will also be deleted automatically. However, updates or modifications made to the Customer entity will not be cascaded to the associated Order entities.

6. CascadeType.DETACH

CascadeType.DETACH is a cascading type in Hibernate that specifies that the detach operation should be cascaded from the parent entity to the child entities.

This means that the child entities will become detached from the persistence context and their state will no longer be managed by Hibernate.

@OneToMany(mappedBy="customer", cascade=CascadeType.DETACH)

private Set<Order> orders;

This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that the detach operation should be cascaded to the associated orders. This means that when a Customer entity is detached, any associated Order entities will also be detached automatically. The Order entities will become detached from the persistence context and their state will no longer be managed by Hibernate.

7. CascadeType.REPLICATE

CascadeType.REPLICATE is a cascading type in Hibernate that specifies that the replicate operation should be cascaded from the parent entity to the child entities.

This means that new child entities will be created and persisted in the database with the same state as the original child entities.

For example, consider a scenario where you have a Customer entity with a one-to-many relationship to Order entities. By using CascadeType.REPLICATE, any associated Order entities will be automatically replicated when the Customer entity is replicated.

Here’s an example of how CascadeType.REPLICATE can be used:

@OneToMany(mappedBy="customer", cascade=CascadeType.REPLICATE)

private Set<Order> orders;

This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that the replicate operation should be cascaded to the associated orders. This means that when a Customer entity is replicated, new Order entities will be created and persisted in the database with the same state as the original Order entities. However, any changes made to the original Order entities after the replication operation will not be replicated to the new Order entities.

8. CascadeType.SAVE\_UPDATE

CascadeType.SAVE\_UPDATE is a cascading type in Hibernate that specifies that the save or update operation should be cascaded from the parent entity to the child entities.

This means that any changes made to the child entities will be persisted in the database along with the parent entity.

@OneToMany(mappedBy="customer", cascade=CascadeType.SAVE\_UPDATE)

private Set<Order> orders;

This code defines a one-to-many relationship between a Customer entity and an Order entity and specifies that the save or update operation should be cascaded to the associated orders. This means that when a Customer entity is saved or updated, any associated Order entities will also be saved or updated automatically. Any changes made to the Order entities will be persisted in the database along with the Customer entity.

# **Hibernate Inheritance Mapping Tutorial**

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

Table Per Hierarchy

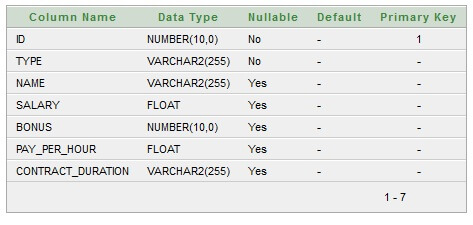
Table Per Concrete class

Table Per Subclass

# **Hibernate Table Per Hierarchy using Annotation**

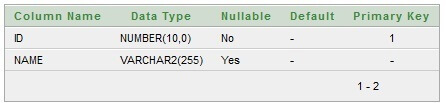
The @Inheritance annotation is used to specify the inheritance strategy for an entity hierarchy. In your case, the strategy InheritanceType.SINGLE\_TABLE is specified. This strategy maps the entire entity hierarchy to a single database table, using a discriminator column to differentiate between different types of entities within the table.

The @DiscriminatorColumn annotation is used to specify the details of the discriminator column for the single table inheritance strategy. In your example, the discriminator column is named "type" and its discriminator type is set to DiscriminatorType.STRING, indicating that the discriminator value will be stored as a string.

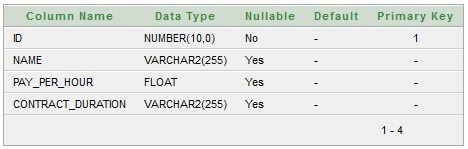


1. @Entity
2. @Table(name = "employee101")
3. @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)
4. @DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
5. @DiscriminatorValue(value="employee")
7. **public** **class** Employee {
8. @Id
9. @GeneratedValue(strategy=GenerationType.AUTO)
11. @Column(name = "id")
12. **private** **int** id;
14. @Column(name = "name")
15. **private** String name;
17. //setters and getters
18. }
19. @Entity
20. @DiscriminatorValue("regularemployee")
21. **public** **class** Regular\_Employee **extends** Employee{
23. @Column(name="salary")
24. **private** **float** salary;
26. @Column(name="bonus")
27. **private** **int** bonus;
29. //setters and getters
30. }
31. @Entity
32. @DiscriminatorValue("contractemployee")
33. **public** **class** Contract\_Employee **extends** Employee{
35. @Column(name="pay\_per\_hour")
36. **private** **float** pay\_per\_hour;
38. @Column(name="contract\_duration")
39. **private** String contract\_duration;
41. //setters and getters
42. }

# **Table Per Concrete class using Annotation**







**@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.

**@AttributeOverrides** defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

1. @Entity
2. @Table(name = "employee102")
3. @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)
5. **public** **class** Employee {
6. @Id
7. @GeneratedValue(strategy=GenerationType.AUTO)
9. @Column(name = "id")
10. **private** **int** id;
12. @Column(name = "name")
13. **private** String name;
15. //setters and getters
16. }

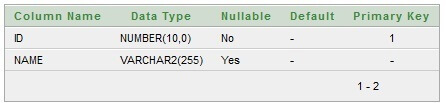
@Entity

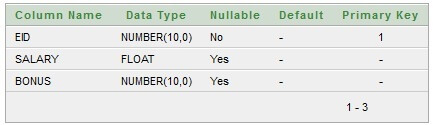
1. @Table(name="regularemployee102")
2. @AttributeOverrides({
3. @AttributeOverride(name="id", column=@Column(name="id")),
4. @AttributeOverride(name="name", column=@Column(name="name"))
5. })
6. **public** **class** Regular\_Employee **extends** Employee{
8. @Column(name="salary")
9. **private** **float** salary;
11. @Column(name="bonus")
12. **private** **int** bonus;
14. //setters and getters
15. }
16. @Entity
17. @Table(name="contractemployee102")
18. @AttributeOverrides({
19. @AttributeOverride(name="id", column=@Column(name="id")),
20. @AttributeOverride(name="name", column=@Column(name="name"))
21. })
22. **public** **class** Contract\_Employee **extends** Employee{
24. @Column(name="pay\_per\_hour")
25. **private** **float** pay\_per\_hour;
27. @Column(name="contract\_duration")
28. **private** String contract\_duration;
30. **public** **float** getPay\_per\_hour() {
31. **return** pay\_per\_hour;
32. }
33. **public** **void** setPay\_per\_hour(**float** payPerHour) {
34. pay\_per\_hour = payPerHour;
35. }
36. **public** String getContract\_duration() {
37. **return** contract\_duration;
38. }
39. **public** **void** setContract\_duration(String contractDuration) {
40. contract\_duration = contractDuration;
41. }
42. }

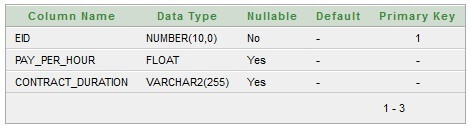
# **Table Per Subclass using Annotation**

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.







1. @Entity
2. @Table(name = "employee103")
3. @Inheritance(strategy=InheritanceType.JOINED)
5. **public** **class** Employee {
6. @Id
7. @GeneratedValue(strategy=GenerationType.AUTO)
9. @Column(name = "id")
10. **private** **int** id;
12. @Column(name = "name")
13. **private** String name;
15. //setters and getters
16. }
17. @Entity
18. @Table(name="regularemployee103")
19. @PrimaryKeyJoinColumn(name="ID")
20. **public** **class** Regular\_Employee **extends** Employee{
22. @Column(name="salary")
23. **private** **float** salary;
25. @Column(name="bonus")
26. **private** **int** bonus;
28. //setters and getters
29. }
30. @Entity
31. @Table(name="contractemployee103")
32. @PrimaryKeyJoinColumn(name="ID")
33. **public** **class** Contract\_Employee **extends** Employee{
35. @Column(name="pay\_per\_hour")
36. **private** **float** pay\_per\_hour;
38. @Column(name="contract\_duration")
39. **private** String contract\_duration;
41. //setters and getters
42. }

# **Hibernate One to One Example using Annotation**

Here, we are going to perform one to one mapping by one-to-one element using annotation. In such case, no foreign key is created in the primary table.

In this example, one employee can have one address and one address belongs to one employee only. Here, we are using bidirectional association. Let's look at the persistent classes.

1. @Entity
2. @Table(name="emp220")
3. **public** **class** Employee {
5. @Id
6. @GeneratedValue(strategy=GenerationType.AUTO)
7. @PrimaryKeyJoinColumn
8. **private** **int** employeeId;
9. **private** String name,email;
10. @OneToOne(targetEntity=Address.**class**,cascade=CascadeType.ALL)
11. **private** Address address;
12. @Entity
13. @Table(name="address220")
14. **public** **class** Address {
16. @Id
17. @GeneratedValue(strategy=GenerationType.AUTO)
18. **private** **int** addressId;
19. **private** String addressLine1,city,state,country;
20. **private** **int** pincode;
22. @OneToOne(targetEntity=Employee.**class**)
23. **private** Employee employee;