### Entity Mapping with Hibernate (`@Entity`, `@Table`, etc.)

In Hibernate, entity mapping refers to how Java classes are mapped to database tables. Hibernate provides annotations that allow you to define this mapping at a high level, abstracting the SQL that would otherwise be necessary to describe these relationships.

Below is a breakdown of the most commonly used annotations in Hibernate for entity mapping.

---

### 1. \*\*`@Entity` Annotation\*\*

The `@Entity` annotation is the core of Hibernate ORM. It marks a class as an entity that should be mapped to a database table.

#### Example:

```java

import javax.persistence.Entity;

@Entity

public class User {

// Fields and methods

}

```

- \*\*`@Entity`\*\*: Marks the class as an entity to be persisted in the database. Without this annotation, Hibernate will not recognize the class for ORM.

---

### 2. \*\*`@Table` Annotation\*\*

The `@Table` annotation is used to specify the table name in the database. If not provided, Hibernate will use the class name as the table name.

#### Example:

```java

import javax.persistence.Entity;

import javax.persistence.Table;

@Entity

@Table(name = "users")

public class User {

// Fields and methods

}

```

- \*\*`@Table(name = "users")`\*\*: Maps the `User` class to the "users" table in the database.

---

### 3. \*\*`@Id` and `@GeneratedValue` Annotations\*\*

The `@Id` annotation marks a field as the primary key, while the `@GeneratedValue` annotation specifies how the primary key should be generated (e.g., auto-increment).

#### Example:

```java

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

// Getters and setters

}

```

- \*\*`@Id`\*\*: Marks the `id` field as the primary key.

- \*\*`@GeneratedValue(strategy = GenerationType.IDENTITY)`\*\*: Specifies that the `id` field is auto-incremented by the database.

Common strategies for `@GeneratedValue` include:

- \*\*`IDENTITY`\*\*: The database generates the primary key.

- \*\*`SEQUENCE`\*\*: Uses a database sequence.

- \*\*`AUTO`\*\*: Hibernate selects the generation strategy based on the database dialect.

---

### 4. \*\*`@Column` Annotation\*\*

The `@Column` annotation allows you to specify the column name in the table, along with constraints like `nullable`, `length`, and `unique`.

#### Example:

```java

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class User {

@Id

private Long id;

@Column(name = "username", nullable = false, unique = true)

private String name;

@Column(name = "email", nullable = false, length = 150)

private String email;

// Getters and setters

}

```

- \*\*`@Column(name = "username")`\*\*: Maps the `name` field to the "username" column.

- \*\*`nullable = false`\*\*: Specifies that the column cannot be `NULL`.

- \*\*`unique = true`\*\*: Enforces a unique constraint on the column.

- \*\*`length = 150`\*\*: Limits the column size to 150 characters.

---

### 5. \*\*`@Temporal` Annotation\*\*

The `@Temporal` annotation is used to map date and time fields in Java to corresponding SQL types like `DATE`, `TIME`, or `TIMESTAMP`.

#### Example:

```java

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Temporal;

import javax.persistence.TemporalType;

import java.util.Date;

@Entity

public class Event {

@Id

private Long id;

@Temporal(TemporalType.DATE)

private Date eventDate;

@Temporal(TemporalType.TIMESTAMP)

private Date eventTimestamp;

// Getters and setters

}

```

- \*\*`@Temporal(TemporalType.DATE)`\*\*: Maps the `eventDate` field to a SQL `DATE`.

- \*\*`@Temporal(TemporalType.TIMESTAMP)`\*\*: Maps the `eventTimestamp` field to a SQL `TIMESTAMP`.

---

### 6. \*\*`@Enumerated` Annotation\*\*

The `@Enumerated` annotation is used to map Java enums to database columns. You can store the enum as either an \*\*ordinal\*\* or a \*\*string\*\*.

#### Example:

```java

import javax.persistence.Entity;

import javax.persistence.Enumerated;

import javax.persistence.EnumType;

import javax.persistence.Id;

@Entity

public class Task {

public enum Status {

PENDING, IN\_PROGRESS, COMPLETED

}

@Id

private Long id;

@Enumerated(EnumType.STRING)

private Status status;

// Getters and setters

}

```

- \*\*`@Enumerated(EnumType.STRING)`\*\*: Maps the enum as a string (`PENDING`, `IN\_PROGRESS`, `COMPLETED`).

- \*\*`EnumType.ORDINAL`\*\*: Maps the enum as an integer (0, 1, 2). This is the default.

---

### 7. \*\*`@OneToOne` Annotation\*\*

The `@OneToOne` annotation maps a one-to-one relationship between two entities.

#### Example:

```java

import javax.persistence.\*;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "address\_id", referencedColumnName = "id")

private Address address;

// Getters and setters

}

```

```java

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Address {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String street;

private String city;

// Getters and setters

}

```

- \*\*`@OneToOne(cascade = CascadeType.ALL)`\*\*: Specifies a one-to-one relationship with cascading.

- \*\*`@JoinColumn(name = "address\_id", referencedColumnName = "id")`\*\*: Specifies that the `address\_id` column in the `User` table references the `id` column in the `Address` table.

---

### 8. \*\*`@ManyToOne` and `@OneToMany` Annotations\*\*

The `@ManyToOne` and `@OneToMany` annotations map a many-to-one relationship, which is the most common relationship in databases.

#### Example of `@ManyToOne` (User and Post):

```java

import javax.persistence.\*;

@Entity

public class Post {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String content;

@ManyToOne

@JoinColumn(name = "user\_id", referencedColumnName = "id")

private User user;

// Getters and setters

}

```

- \*\*`@ManyToOne`\*\*: Specifies a many-to-one relationship, meaning many `Post` entities can be associated with one `User`.

- \*\*`@JoinColumn(name = "user\_id")`\*\*: Specifies that the foreign key column `user\_id` references the primary key `id` in the `User` entity.

---

### 9. \*\*`@OneToMany` Annotation\*\*

The `@OneToMany` annotation is used to map a one-to-many relationship, where one entity is associated with multiple child entities.

#### Example of `@OneToMany` (User and Post):

```java

import javax.persistence.\*;

import java.util.List;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Post> posts;

// Getters and setters

}

```

- \*\*`@OneToMany(mappedBy = "user")`\*\*: Maps the `posts` list in the `User` class to the `user` field in the `Post` class.

- \*\*`cascade = CascadeType.ALL`\*\*: Specifies that operations (such as persist, remove) on the `User` entity cascade to the related `Post` entities.

- \*\*`fetch = FetchType.LAZY`\*\*: Specifies lazy loading, meaning the `posts` will not be loaded until explicitly accessed.

---

### 10. \*\*`@ManyToMany` Annotation\*\*

The `@ManyToMany` annotation maps a many-to-many relationship between two entities. In the database, this is typically represented by a join table.

#### Example of `@ManyToMany` (Student and Course):

```java

import javax.persistence.\*;

import java.util.Set;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private Set<Course> courses;

// Getters and setters

}

```

```java

import javax.persistence.\*;

import java.util.Set;

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

@ManyToMany(mappedBy = "courses")

private Set<Student> students;

// Getters and setters

}

```

- \*\*`@ManyToMany`\*\*: Defines a many-to-many relationship.

- \*\*`@JoinTable`\*\*: Specifies the join table that connects `Student` and `Course` entities.

- \*\*`joinColumns`\*\*: Maps the foreign key for the `Student`.

- \*\*`inverseJoinColumns`\*\*: Maps the foreign key for the `Course`.

---

### Conclusion

Entity mapping in Hibernate is the process of linking Java classes and fields to database tables and columns. By using annotations such as `@Entity`, `@Table`, `@OneToMany`, and others, Hibernate provides a powerful and flexible way to define how data is stored and retrieved from a relational database.