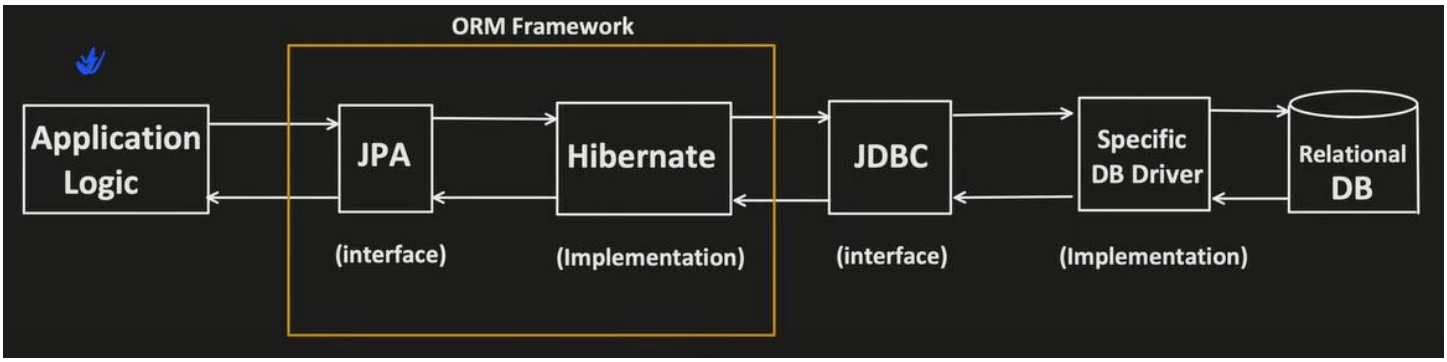
JDBC | JdbcTemplate | ORM | Hibernate



JDBC (Java Database Connectivity)

- It's part of the **java.sql** package and enables Java applications to **interact with databases.**
- **JDBC** is a **standard Java API** that provides interfaces and classes to:
    - Establish a connection with DB.
    - Execute SQL queries (e.g., SELECT, INSERT, UPDATE, DELETE)
    - Process the Results (via ResultSet)
- Actual implementation is provided by Specific DB Drivers.
- JDBC uses **drivers** to communicate with the database. Drivers are divided into 4 Types.
    1. JDBC-ODBC Bridge Driver
    2. Native-API Driver
    3. Network Protocol Driver
    4. Thin Driver

DriverManager

- A core class in java.sql package.
- Manages a list of registered database drivers.
- Responsible for **establishing the connection** between Java app and the DB.
- It is responsible to register and unregister Database Drivers.

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
DriverManager.deregisterDriver(driver); // Also valid
```

- The **getConnection()** method returns a **Connection object** to interact with the Database.

```
String url = "jdbc:mysql://localhost:3306/mydb";
String username = "root";
String password = "root";
Connection con = DriverManager.getConnection(url, username, password);
```

DB-specific JDBC drivers:

- Each of driver having implementation of JDBC Api.

Database	Driver Class Name	JDBC URL format
MySQL	com.mysql.**cj**.jdbc.Driver	jdbc:mysql://host:port/dbname
PostgreSQL	org.**postgresql**.Driver	jdbc:postgresql://host:port/dbname
Oracle	oracle.jdbc.**OracleDriver**	jdbc:oracle:thin:@host:port:SID
SQL Server	com.microsoft.sqlserver.jdbc .**SQLServerDriver**	jdbc:sqlserver://host:port;databaseName= dbname
H2 (in-memory)	org.**h2**.Driver	jdbc:h2:mem:testdb

JDBC Interaction Steps:

1. Import JDBC packages.
2. Register the JDBC driver.

   Class.forName("com.mysql.cj.jdbc.Driver"); // throw ClassNotFoundException
3. Establish a connection.

   Connection conn = DriverManager.getConnection(url, user, password); // throw SQLException
4. Create a Statement or PreparedStatement.

   Statement stmt = conn.createStatement(); // can throw SQLException

   PreparedStatement pstmt = conn.prepareStatement(sql);
5. Execute SQL queries (query or update).

   ResultSet rs = stmt.executeQuery(query); // select

   int rowsAffected = stmt.executeUpdate(query) // insert, update, delete
6. Process the ResultSet (if applicable).

   while (rs.next()) { }
7. Handling exceptions using try-catch.

   ClassNotFoundException and SQLException
8. Close all resources to avoid memory leaks.

   if (rs != null) rs.close();

   if (stmt != null) stmt.close();

   if (conn != null) conn.close();

Transaction Management

- Manage transactions by setting auto-commit to false, and explicitly commit or rollback changes.

```
conn.setAutoCommit(false); // Start transaction
conn.commit();             // Commit transaction
conn.rollback();           // In case of error, rollback
```

Methods:

1. executeQuery()
   - Use this Method for **Select** Operations.
   - Return a Group of Records, which are represented by ResultSet Object.

   Eg: ResultSet rs = statement.executeQuery("select * from movies");

2. executeUpdate()
   - Use this Method for **Non-Select** Operations (Insert|Delete|Update).
   - Return a Numeric Value represents the Number of Rows effected.

   ```
   int rowCount
           = statement.executeUpdate("delete from employees where esal>100000");
   ```

3. execute()
   - Use this Method for **both Select and Non-Select** Operations.

   Eg: boolean flag = statement.execute("Dynamic query");

Statement(I)

|

PreparedStatement(I) :

- The main advantage of PreparedStatement is the query will be compiled only once even though we are executing multiple times.

```
String sqlQuery = insert into employees values(?, ?, ?, ?);
PreparedStatment pst=con.prepareStatement(sqlQuery);
pst.setInt(1,100);
pst.setString(2,"durga");
pst.setDouble(3,1000);
pst.setString(4,"Hyd");
```

➔ Spring is **not** providing **its own ORM** framework but it's provide **Spring ORM, Spring Data JPA** modules.

- Hibernate provide abstraction on JDBC
- Spring ORM, Spring Data JPA provide abstraction on Hibernate.
- By default, Spring ORM, Spring Data JPA provides abstraction on JDBC

Using JDBC without SpringBoot

```java
public class DatabaseConnection {

    public Connection getConnection() {
        try {
            // H2 Driver loading
            Class.forName( className: "org.h2.Driver");

            // Establish connection with DB
            return DriverManager.getConnection( url: "jdbc:h2:mem:userDB", user: "sa", password: "");

        }
        catch (ClassNotFoundException | SQLException e) {
            //handle  exception
        }

        return null;
    }
}
```

DB Name

```java
public void createUserTable() {
    try {
        Connection connection = new DatabaseConnection().getConnection();
        Statement statementQuery = connection.createStatement();
        String sql = "CREATE TABLE users(user_id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(100), age INT)";
        statementQuery.executeUpdate(sql);
    }
    catch (SQLException e) { /** handle exception **/ }
    finally { /** close statementQuery and db connection **/ }
}

public void createUser(String userName, int userAge) {
    try {
        Connection connection = new DatabaseConnection().getConnection();
        String sqlQuery = "INSERT INTO users(user_name, age) VALUES (?, ?)";
        PreparedStatement preparedQuery = connection.prepareStatement(sqlQuery);
        preparedQuery.setString( parameterIndex: 1, userName);
        preparedQuery.setInt( parameterIndex: 2, userAge);
        preparedQuery.executeUpdate();
    }
    catch (SQLException e) { /** handle exception **/ }
    finally { /** close preparedQuery and db connection **/ }
}
```

```java
public void readUsers() {
    try {
        Connection connection = new DatabaseConnection().getConnection();
        String sqlQuery = "SELECT * FROM users";
        PreparedStatement preparedQuery = connection.prepareStatement(sqlQuery);
        ResultSet output = preparedQuery.executeQuery();
        while (output.next()) {
            String userDetails = output.getInt( columnLabel: "user_id") +
                    ":" + output.getString( columnLabel: "user_name") +
                    ":" + output.getInt( columnLabel: "age");
            System.out.println(userDetails);
        }
    }
    catch (SQLException e) { /** handle exception **/ }
    finally { /** close preparedQuery and db connection **/ }
}
```

- But there are so much of repeated BOILERCODE present like:
  - Driver class loading
  - DB Connection Making
  - Exception Handling
  - Closing of the DB connection and other objects like Statement etc.
  - Manual handling of DB Connection Pool etc.

Issues with JDBC API

- JDBC requires a lot of **Repetitive code for establishing connections**, Creating Statement or PreparedStatement, executing queries, handling exceptions, and Closing resources.
- The developer must **Explicitly Close Connection**, **Statement**, and **ResultSet**, which increases the **risk of resource leaks** if not handled properly.
- JDBC **throws checked exceptions** (SQLException) for every DB operation, which must be handled explicitly.
- JDBC **works directly with Rows and Columns (**ResultSet**),** making it **hard to map relational data to Java objects (POJOs).**
- JDBC does **not include built-in Connection Pooling**. **Without a connection pool**, **opening and closing database connections** for each request is **slow** and **inefficient**.
- JDBC code **tightly coupled with DB**, If the database changes, the SQL queries and driver settings must be updated.

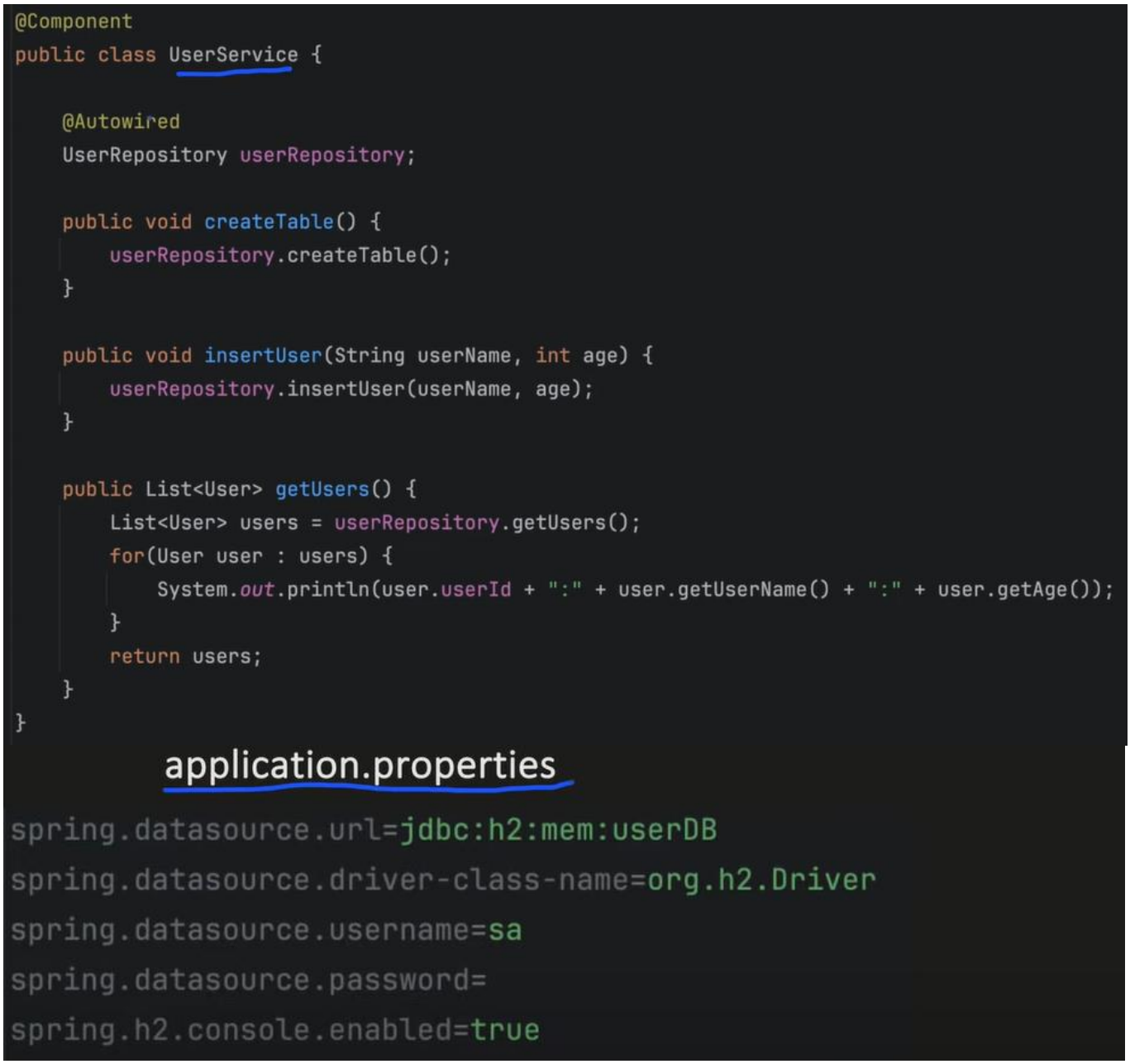
Using JDBC with SpringBoot (JdbcTemplate)

pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

- Springboot provides JdbcTemplate class, which helps to remove all the boiler code.

```java
@Repository
public class UserRepository {

    @Autowired
    JdbcTemplate jdbcTemplate;

    public void createTable() {
        jdbcTemplate.execute( sql: "CREATE TABLE users (user_id INT AUTO_INCREMENT PRIMARY KEY, " +
                "user_name VARCHAR(100), age INT)");
    }

    public void insertUser(String name, int age) {
        String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?)";
        jdbcTemplate.update(insertQuery, name, age);
    }
    public List<User> getUsers() {
        String selectQuery = "SELECT * FROM users";
        return jdbcTemplate.query(selectQuery, (rs, rowNum) -> {
            User user = new User();
            user.setUserId(rs.getInt( columnLabel: "user_id"));
            user.setUserName(rs.getString( columnLabel: "user_name"));
            user.setAge(rs.getInt( columnLabel: "age"));
            return user;
        });
    }
}
```



```java
@Component
public class UserService {

    @Autowired
    UserRepository userRepository;

    public void createTable() {
        userRepository.createTable();
    }

    public void insertUser(String userName, int age) {
        userRepository.insertUser(userName, age);
    }

    public List<User> getUsers() {
        List<User> users = userRepository.getUsers();
        for(User user : users) {
            System.out.println(user.userId + ":" + user.getUserName() + ":" + user.getAge());
        }
        return users;
    }
}
```

application.properties

```
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
```

-Or-

```
getDataSource() {

        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        ds.setUsername("root");
        ds.setPassword("Mysqsk45@123");
        return DS;
}
public JdbcTemplate getJdbcTemplate() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(getDataSource());
}
```

- Driver class loading
    - JdbcTemplate load it at the time of application startup in DriverManager class.
- DB Connection Making
    - jdbcTemplate takes care of it, whenever we execute any query.
- Exception Handling
    - in Plain JDBC, we get very abstracted 'SQLException' but in jdbcTemplate, we get granular error like DuplicateKeyException, QueryTimeoutException etc.. (defined in org.springframework.dao package).
- Closing of the DB connection and other resources
    - when we invoke update or query method, after success or failure of the operation, jdbcTemplate takes care of either closing or return the connection to Pool itself.
- Manual handling of DB Connection Pool
    - Springboot provides default JDBC connection pool i.e. 'HikariCP' with Min and Max pool size of 10. And we can change the configuration in 'application.properties'.

        spring.datasource.hikari.maximum-pool-size=10

        spring.datasource.hikari.minimum-idle=5

- We can also configure different JDBC connection pool (instead of Hikari) if we want like below without provide in .properties file:

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
}
```

JdbcTemplate Methods

- **update(String sql, Object... args)**
  - Used for: Insert, Update, Delete

```
String insert Query = "INSERT INTO users (user_name, age) VALUES (?, ?)";
int rowsAffected = jdbcTemplate.update(insertQuery, "X", 27);

String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?";
int rowsAffected = jdbcTemplate.update(updateQuery, 29, 1);

String deleteQuery= "DELETE FROM users WHERE user_id = ?";
int rowsAffected = jdbcTemplate.update(deleteQuery, 1);
```

- **update(String sql, Object... args)**
  - Used for: Insert, Update, Delete

```
String insertQuery= "INSERT INTO users (user_name, age) VALUES (?, ?)";
jdbcTemplate.update(insertQuery, (PreparedStatement ps) -> {
        ps.setString(1, "X");
        ps.setInt(2, 25);
});

String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?";
jdbcTemplate.update(updateQuery, (PreparedStatement ps) -> {
        ps.setString(1, 29);
        ps.setInt(2, 1);
});

String deleteQuery= "DELETE FROM users WHERE user_id = ?";
jdbcTemplate.update(deleteQuery, (PreparedStatement ps) -> {
        ps.setInt(1, 2);
});
```

- **query(String sql, RowMapper<T> rowMapper)**
  - Used to get multiple rows

```
List<User> users = jdbcTemplate.query "SELECT * FROM users". (rs, rowNum) -> {
    User user = new User();
    user.setUserId(rs.getInt("user_id"));
    user.setUserName(rs.getString("user_name"));
    user.setAge(rs.getInt("age"));
    return user;
});
```

- **queryForList(String sql, Class<T> elementType)**
  - Used to get single column of multiple rows

    List<String> userNames = jdbcTemplate.queryForList("SELECT user_name FROM users", String.class);
- **queryForObject(String sql, Object[] args, Class<T> requiredType)**
  - Used to get single row

    User user = jdbcTemplate.queryForObject("SELECT * FROM users WHERE user_id = ?", new Object[]{1},

    User.class);
- **queryForObject(String sql, Class<T> required Type)**
  - Used to get single value

    int userCount = jdbcTemplate.queryForObject("SELECT COUNT(*) FROM users", Integer.class);

ORM

- Object-oriented programming (OOP) allows us to represent **real-world entities as objects** and their relationships.
- Objects contain data (**attributes**) and code (**methods**) Then
- how to persist the objects' data using non-object-oriented databases, like **RDBMS**.
- An Object-Relational Mapping tool, **ORM**, is a framework used to map the **Objects** to the Data stored in the Database **table rows (Mapping** between **Java Class (**Entities) **and Database Tables)**
- A Concept that allows to use Java **objects to interact with the DB** without SQL queries.
- Every ORM is having ability to generate Schemas dynamically based on JPA Annotations.
- Classes that deal with object-relational mappings are called Models.
- Simplifies the Data Creation, Data Manipulation and Data Access.

Ex:

```
public class Employee {
        private int id;
        private String first_name;
        private String last_name;
        private int salary;
}
create table EMPLOYEE (
        id INT NOT NULL auto_increment,
        first_name VARCHAR(20) default NULL,
        last_name VARCHAR(20) default NULL,
        salary INT default NULL,
        PRIMARY KEY (id)
);
```

Why ORM

➢ **Integrating the OOPs Concept with RDBMS** concept has **technical difficulties** as follows:

1. **Mismatch** between the **number of classes in Object Model** and the **number of tables in Relational Model** called **Granularity Problem**.
   Ex. Student with Address class and one Student table
2. In **Object Model** we **have Inheritance**, but **don't** have **in Relational Model.**
3. In Object Model, the **association** is **represented** using **reference variables**, but **in Relational Model**, it's **represent** using **Foreign Keys**.

What is object/relational mapping metadata?

- **ORM** tools require a metadata format for the application to specify the mapping between classes and tables, properties and columns, associations and foreign keys, Java types and SQL types. This information is called the **object/relational mapping metadata**.
- It defines the transformation between the different data type systems and relationship representations**.**

→ **ORM Framework** => Hibernate, EclipseLink, OpenJPA

JPA

What is JPA? (Java Persistence API)

- The java persistence API provides a specification for persisting, reading, and managing data from your java object to your relational tables in the database.
- Its **Specification** provides certain **Rules and Guidelines** to develop **Standard ORM tools/ Frameworks**.
- The **javax.persistence** package contains Annotations and JPA classes and interfaces. eg. Hibernate, EclipseLink
- It just says "Here's how you should handle saving, updating, deleting, and querying data in a database using Java classes."

Feature / Aspect	JDBC	Spring JDBC	JPA / Hibernate

Level of Abstraction	Low (very manual)	Medium (simplifies JDBC)	High (full ORM)

Code Complexity / Boilerplate	High (manual connection, query, mapping)	Medium (less boilerplate with template)	Low (minimal with repositories & annotations)

Object Mapping (ORM)	❌ No ORM support	❌ No ORM	✅ Yes (Entity mapping, relationships)

Query Language	SQL	SQL	JPQL / HQL / Criteria API / Native SQL

Transaction Management	Manual or via JDBC	Declarative via Spring	Declarative via Spring or JTA

Connection Management	Manual	Auto via Spring's JdbcTemplate	Auto via EntityManager + Spring

Exception Handling	Checked (SQLException)	Converts to `DataAccessException`	Runtime exceptions (e.g. `PersistenceException`)

Mapping ResultSet to POJO	Manual (`rs.getXyz()`)	Manual, but simplified	Auto via annotations

Relationships (OneToMany, etc.)	❌ Not supported	❌ Not supported	✅ Supported via annotations

Caching Support	❌ None	❌ None	✅ First- and Second-level caching

Lazy Loading / Eager Fetching	❌ Not available	❌ Not available	✅ Fully supported

Pagination Support	Manual (`LIMIT`, `OFFSET`)	Manual	✅ Built-in via `Pageable`, `Query.setMaxResults()`

Best For	Low-level control, simple apps	Mid-sized apps needing some abstraction	Complex apps with rich domain models

Learning Curve	Low–Medium	Medium	Medium–High (but cleaner long term)

HIBERNATE

1. **Data Persistence**
2. **Introduction**
3. **Architecture**
4. **JPA vs Hibernate**
5. **XML vs Annotation**
6. **Entity Lifecycle**
7. **CRUD Operations**
8. **Hibernate Caching**
9. **Id Generators**
10. **Composite Mapping**
11. **Hibernate Inheritance**
12. **Entity State**
13. **Hibernate Relationships/Association**
14. **Pagination**
15. **Filters**
16. **Hibernate Locking**
17. **Transaction**

Data Persistence

- Permanent Storage of Data
- Core Component of Data Persistence

1. Data: WHAT to persist?
    - Primitive data and Java Objects
2. Medium: HOW to persist?
    - Java I/O stream
    - Serialization
    - JDBC API
3. Storage: WHERE to persist?
    - File
    - Database

Technologies to develop persistent logic

1. JDBC
2. Spring JDBC
3. ORM Framework
4. Spring Data JPA etc.

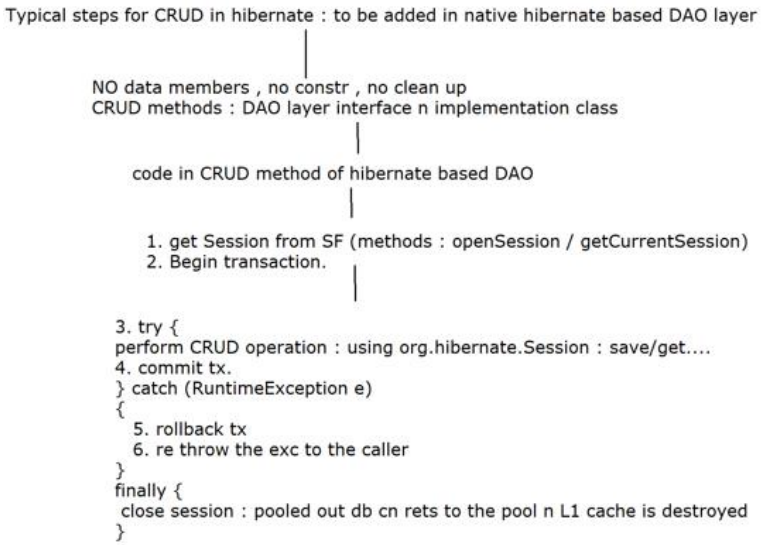
- **Best practices to Develop Persistent logic**
  1. Maintain logic as separate layer (**DAO/Repository**)
  2. Should follows **Table per Dao/Repository**
  3. Creating Table name, Column name, sequence the **length limitation** is **30 characters.**
  4. **If** required name having **more than 30** characters then **ignore vowels**(a,e,i,o,u) from name.
  5. For **every table** at least **one Primary Key** is Highly recommended.
  6. For **every table** to maintain below **Auditing columns is Highly recommended.**
     - CREATED_DATE
     - CREATED_BY
     - UPDATED_DATE
     - UPDATED_BY
  7. for **every Primary Key** create **dedicated sequence.**
  8. **Always** recommended to **maintain Cache for static table data** to avoid DB interaction.

Introduction

- Hibernate is a **Java-based ORM** (Object/relational mapping) tool.
- Hibernate implements the specifications of JPA (Java Persistence API) for data persistence and provides extra features.
- A **tool** or **library** that **follows the JPA rules** and does the real work.
- Hibernate takes your Java objects and maps them to database tables.
- It provides extra features on top of JPA too (like caching, lazy loading, etc.).

Advantages of Hibernates

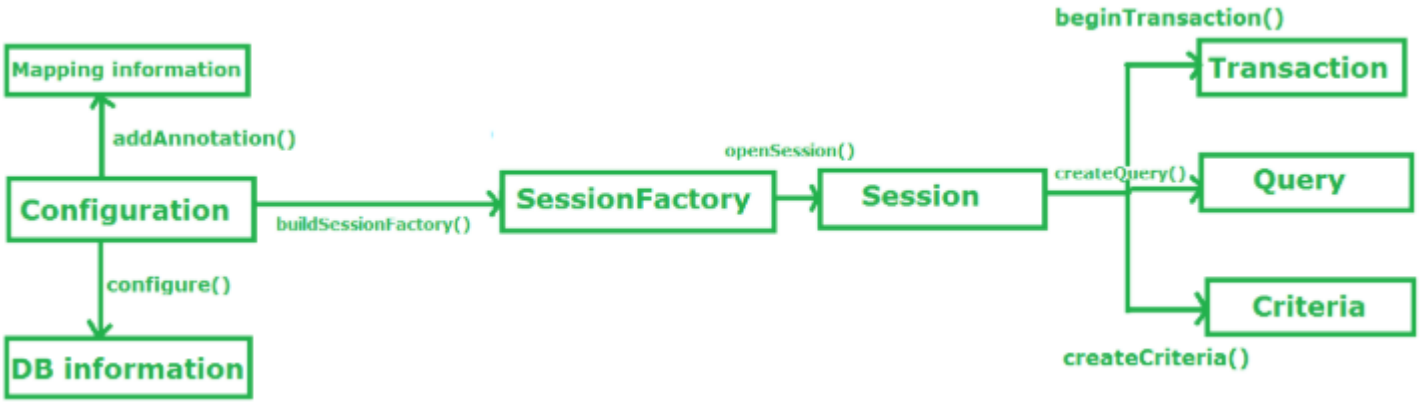
- Hibernate is Open Source, Lightweight, ORM tool.
- Fast Performance: It uses Cache internally. **First Level cache** (by default enabled) and **Second Level cache**
- Generates Database Independent Query: Can perform Database **CRUD Operations without** writing **raw SQL Query.**
- Hibernate **supports different databases** (eg. MySQL, PostgreSQL, Oracle).
- Automatic Table Creation: There is **no need to Create Tables** in Database **Manually**.
- **Fetching data from Multiple Tables** is **Easy** in hibernate framework.
- It **handles relationships between Entities** effectively. (eg. **one-to-one, one-to-many**)
- Hibernate simplifies **handling inheritance, polymorphism, and associations in Database Schemas**. (like @OneToMany and @ManyToMany)
- Configuration can be done by **'.xml' or '.java'**
- Can easily implement **Pagination and Sorting.**



Architecture

Explain Hibernate Architecture?

1. The Hibernate architecture is categorized in **four layers.**
   - Java application layer
   - Hibernate framework layer.
   - Backend API layer
   - Database layer
2. Hibernate framework uses many objects such as **SessionFactory**, **Session**, **Transaction** etc. along with existing Java API such as **JDBC** (Java Database Connectivity), **JTA** (Java Transaction API) and **JNDI** (Java Naming Directory Interface).



1. **Configuration (C)**

- A class in the **org.hibernate.cfg** package responsible for **bootstrapping the Hibernate framework**.
- It is responsible to Loads and parses both **'. CFG.xml'** and **'. HBM.xml'** files and Reads properties like Database connection settings, Dialect, mappings (Entity → Table), and other properties.
- It **Checks** whether the **CFG file is syntactically correct or not.**
- If it's **Not valid** then **throw** Runtime **Exception**.
- If it's **Valid,** then it creates **InMemeory MetaData** and return Configuration Object.
- **buildSessionFactory** () gets the **JDBC information** from **CFG Object** and Establish **Database connection** (via JDBC) and returns a **singleton SessionFactory** instance.

XML Parser

- Hibernate internally uses an **XML parser** (like JAXB, DOM, or SAX) to:
- Load and parse **XML config/mapping** files, checks whether CFG file is well-formed and **Valid** or **not**.
- If it's **valid** and well-formed then **it reads and process** the given **xml** file and **generates InMemeory Meta-Data.**

2. **SessionFactory(I)** (org.hibernate.SessionFactory)

- An Interface present in **org.hibernate** package. It's used to create and manage **Session** objects**.**
- It is **Immutable** and **Thread-Safe** in nature. It can be safely **shared across multiple threads.**
- SessionFactory created **Once per Application** (**Singleton** pattern).
- Created using the **`Configuration.buildSessionFactory()`** method.
- **`openSession()`:**
  - **returns** a **new Session object** (not bound to any Transaction or Context).
  - Should be closed manually.
- **`getCurrentSession()`**
  - Returns a session **bound to the current context**. which is usually **managed by TransactionManager**.
  - Managed **automatically** (auto-close after transaction ends).
  - Requires proper configuration:

  <property name="hibernate.current_session_context_class">**thread**</property>

- **Responsible** to manage **Connection Pooling** (via underlying JDBC or a pool like C3P0, HikariCP)**, First-level Cache** (within each Session)**, Session** and **entity lifecycle,** etc.

3. **Session(I)** (org.hibernate.Session)

- An Interface Present in **org.hibernate** package.
- It **opens the Connection/Session** between the **Application** and **Database** software.
- Used to perform **CRUD operations** (Create, Read, Update, Delete) on database entities.
- Opens a connection to the database (internally via JDBC).
- It is a **Light-weight** object and **not Thread-Safe.**

4. **Transaction**

- Interface in the org.hibernate package.
- Used to **manage atomic units of work** (i.e., DB operations that must succeed or fail together).
- Transactions are important to **ensure data consistency** and prevent partial updates.
- Common methods:
  - begin()
  - commit() – makes changes **permanent**
  - rollback() – **undo** changes if an error occurs

5. **ConnectionProvider**

- An interface present in org.hibernate.connection.ConnectionProvider used to abstract the way Hibernate obtains **JDBC connections**.
- Acts as a **factory for JDBC Connection objects**.
- It Hides details of whether connections come from: java.sql.DriverManager (direct JDBC) or javax.sql.DataSource (connection pools like HikariCP, C3P0).

6. **Query**

- An **interface** in the org.hibernate package.
- Used to represent a **Hibernate Query** written in **HQL (Hibernate Query Language)**.
- A Query instance is obtained by calling Session.createQuery().
- Can perform:
  - Select operations
  - Parameter binding
  - Pagination
  - Bulk updates/deletes

```
Query query = session.createQuery("from Employee where salary > 30000");
```

7. **Criteria**

- **Hibernate Criteria** is a **type-safe**, **object-oriented** API used to build queries dynamically without writing HQL or SQL directly.
- Helps in building queries using Java code (programmatic query building).
- Present in:
  - Legacy API: `org.hibernate.Criteria` (deprecated since Hibernate 5.x)
  - Modern API: `JPA Criteria API (javax.persistence.criteria.CriteriaBuilder)`

```
Session session = sessionFactory.openSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> root = cq.from(Employee.class);

// Add condition: salary > 30000
cq.select(root).where(cb.gt(root.get("salary"), 30000));

List<Employee> results = session.createQuery(cq).getResultList();
```

Configuration cfg=new Configuration().configure (.cfg.xml);

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession(); (or)

{ Session session = sessionFactory.getCurrentSession(); }

Transaction tx=session.beginTransaction();

tx.commit();

What are the Extension interfaces that are there in hibernate?

- **ProxyFactory** interface - used to create proxies
- **ConnectionProvider** interface – used for JDBC connection management
- **TransactionFactory** interface – Used for transaction management
- **Transaction** interface – Used for transaction management
- **TransactionManagementLookup** interface – Used in transaction management.
- **Cahce** interface – provides caching techniques and strategies
- **CacheProvider** interface – same as Cache interface
- **ClassPersister** interface – provides ORM strategies
- **IdentifierGenerator** interface – used for primary key generation
- **Dialect** abstract class – provides SQL support

SF API openSession vs getCurrentSession
(rets Session object)

openSession	getCurrentSession
1. Irrespective of the fact that session exists or doesn't exist, a NEW session object is returned to the caller.	1. If session obj exists , then existing session object is reted , ow. new session is created n reted.
2. Prog MUST explicitely close the session -- session.close() , using finally block.	2. Session is auto closed --upon tx boundary.

What is meant by Method chaining?

it is not mandatory to use this format.

```
SessionFactory sessions = new Configuration()
    .addResource("myinstance/MyConfig.hbm.xml")
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

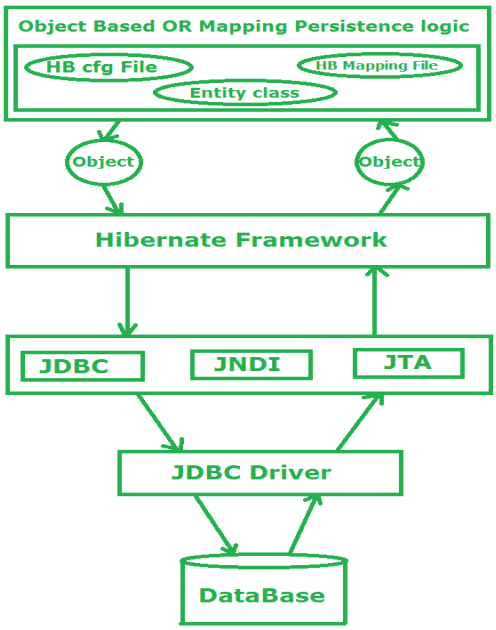


Fig: Working Flow of Hibernate framework to save/retrieve the data from the database in form of Object

JPA vs Hibernate

Feature	JPA	Hibernate
What it is	Specification (interface)	Implementation (actual library)
Does it work alone?	No	Yes
Provides rules for	Object-Relational Mapping (ORM)	ORM + extra features
Popularity	Used via implementations	Most widely used implementation

XML vs Annotation

- Using Hibernate, manually needs to provide **configuration to map the Java classes to database tables**.
- **Configuration can be done** in two ways:
  - XML Mapping
  - Annotations

XML Mapping	Annotations
Separate .hbm.xml file	Inside the Java class (inline)
Clean class code, but mappings in another place	Easy to read, but mixes logic + config
Harder (have to look in two places)	Easier (all in one place)
More flexible in complex scenarios	Good for most common use cases
More setup work	Less setup, faster to write

What are the most common methods of configuring Hibernate?

- **Using hibernate.properties**
  - Place a file named `hibernate.properties /.yml` in the **classpath**.
  - It contains key-value pairs like:

```
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.cj.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/mydb
hibernate.connection.username=root
hibernate.connection.password=root
hibernate.show_sql=true
```

- **Using hibernate.cfg.xml**
  - A **more structured and preferred** method.
  - Also placed in the **classpath**.

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

    <!-- Entity Mapping -->
    <mapping class="com.example.model.Employee"/>
  </session-factory>
</hibernate-configuration>
```

How can the mapping files be configured in Hibernate?

1. **In Java Code**

```
Configuration cfg = new Configuration();

cfg.addAnnotatedClass(Employee.class);

cfg.addResource("employee.hbm.xml"); // If using XML-based mapping
```

2. **In `hibernate.cfg.xml`**

```
<mapping resource="employee.hbm.xml"/>

<mapping class="com.example.model.Employee"/>
```

What happens when both `hibernate.properties` and `hibernate.cfg.xml` are in the classpath?

- If both are present in the classpath, the settings in **hibernate.cfg.xml** will **override** the settings in **hibernate.properties.**

How to set Hibernate to log all generated SQL to the console?

- By setting the **hibernate.show_sql** property to true.

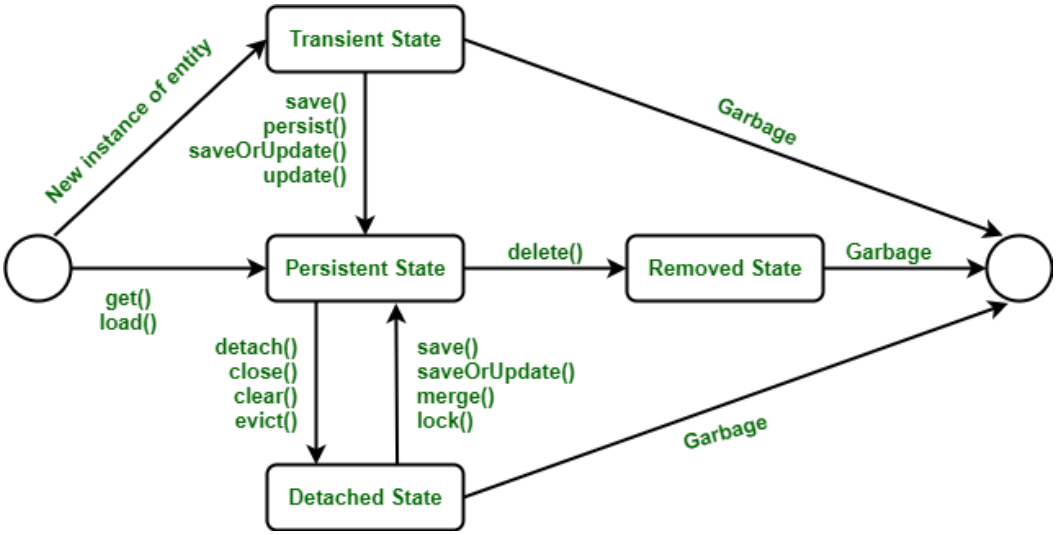
Useful Logging Properties:

hibernate.format_sql=true # Formats the SQL nicely

hibernate.use_sql_comments=true # Adds comments for readability

Entity Lifecycle

- Hibernate Lifecycle nothing but **lifecycle of A Mapped Object of Entity Classes** in hibernate.
- **Each Entity** relates to the **lifecycle** and **Passes** through the **Various Stages** of the lifecycle.
- By creating New Object of Entity can store it into Database, can fetch existing data of an Entity from Database.
- There are four states of the Hibernate Lifecycle:
  1. **Transient State**
  2. **Persistent State**
  3. **Detached State**
  4. **Removed State**



T - P - D - R

➢ Transient State

- When a POJO object is created using the new keyword, it is in the **Transient state**.
- At this point:
  - The object is **not associated** with any Hibernate Session.
  - The object is **not linked** to any row in the database.
  - No persistent identity (**primary key**) is assigned by Hibernate.
- Any Changes to the Object are **not reflected** to the Database Table.
- Transient objects are independent of Hibernate Session, andThe object **exists only in JVM memory (Heap).**

Ex:

```
Employee e = new Employee();      //Transient state
e.setName("John Doe");
e.setFirstName("Neha");
```

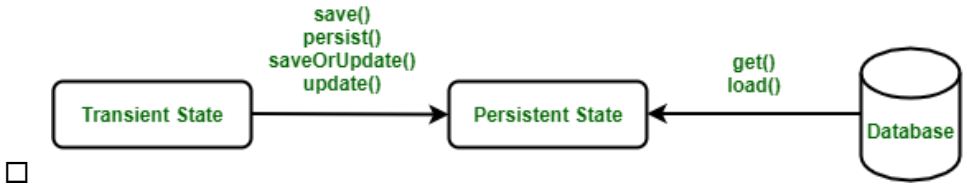
➢ Persistent State

- When a **Hibernate Session** is managing a POJO/entity, the object is in the **Persistent State**.
- Hibernate starts **tracking changes** to the object and **automatically synchronizes** them with the database.
- The object now **represents a specific row** in the database table.
- Any field change to the object will be **reflected in the database** (no need to explicitly call update).
- Two ways from

➢ Transient → Persistent:

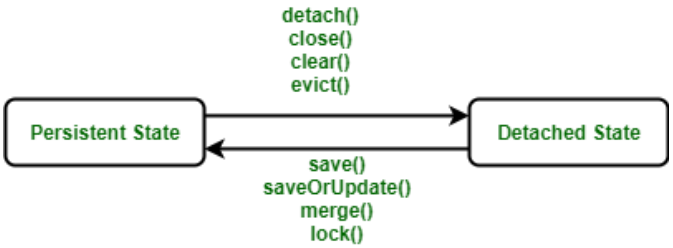
1. **Save a new entity to DB:**
    - session.**save**(entity); // returns generated ID
    - session.**persist**(entity); // void return
    - session.**saveOrUpdate**(entity); // handles both save or update

2. **Load an existing entity from DB:**
    - session.**get**(Employee.class, 1); // Eagerly fetches from DB
    - session.**load**(Employee.class, 1); // Lazy proxy until accessed



➢ Detached → Persistent:

- session.**update**(entity); // Reattaches a detached object
- session.**merge**(entity); // Merges changes from detached into managed instance
- session.**lock**(entity); // Reattaches without checking version (rare use)



➢ Detached State

- The Object comes From Persistent State to Detached State and is **no longer associated** with an active Hibernate Session.
- This happens when:
  - The Session is **closed** close().
  - Or the object is **explicitly removed** from the session cache clear().
- In this state:
  - The object still holds the **database identifier (ID)**.
  - But **Hibernate no longer tracks changes** to it.
  - Any changes made to the object **won't be reflected in the DB** unless re-attached to a session.

➢ **Persistent → Detached:**

- session.close(); // Close the session
- session.clear(); // Clears all persistent objects
- session.evict(employee); // Removes one specific object
- session.detach(employee); // Detaches object (Hibernate 5.2+)

➢ **Detached → Persistent (Reattachment):** reattach to ***new session*** and save the changes:

- session.update(employee); // Reattaches the same instance
- session.merge(employee); // Copies values to a managed object

```
Session session = sessionFactory.openSession();
Employee employee = session.get(Employee.class, 1);  // Persistent

session.close();                // Detached

employee.setName("Updated Name");    // No effect DB unless re-attached

Session newSession = sessionFactory.openSession();
newSession.update(employee);     // Reattached — now in Persistent again
```

- **session.detach()** is newer (Hibernate 5.2+)
- **merge()** creates/returns a **new managed copy**, while **update()** reattaches the **same instance**

➢ Removed State

- An entity is in the **Removed State** when it is **Scheduled** for Deletion using **session.delete(entity)** or **entityManager.remove(entity)**.
- It is still in **Persistent state until the transaction is committed**.
- The actual row is **not deleted from the database immediately but** is **marked for deletion.**
- Once the transaction is **committed**, the corresponding row will be deleted from the database.
- After deletion, the object becomes **Transient** (detached from the **DB** and **session**, **not tracked** anymore).

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Employee employee = session.get(Employee.class, 1);  // Persistent state

session.delete(employee);   // Moved to Removed state (still managed)

tx.commit();                // Now actually deleted from DB → Transient
session.close();
```

State	Transition	Method
Transient	→ Persistent	persist(), save(), saveOrUpdate()
Persistent	→ Detached	detach(), evict(), close(), clear(),
Detached	→ Persistent	merge(e), lock(e), update(),
Persistent	→ Removed	delete()

Hibernate POJO / Entity States

	Is It a Part of L1 Cache ?	Does it have DB Identity
Transient	NO	NO
Persistent	YES	transient-->persistent(save/persist...) -- gains DB identity upon commit. doesn't exist ---persistent (get/load/jpql) -- YES
Detached	NO	YES
Removed	Trigger--session.delete (ref) Marked for removal YES	Upon commit --delete query fired --L1 cache destroyed --not a part of DB or cache

Hibernate Annotations

1. Entity Annotations

@Entity

- Mark a class as JPA entity, meaning this class will **map to Table** in the DB.
- **name**: Specifies the **table name** in the database.

@Table(name="table_name")

- It is used to customize table mapping — table name, schema, and constraints.

➢ **Attribute**

- **name**: Table name in the database.
  Ex. `@Table(name = "EMPLOYEE_TABLE")`

- **uniqueConstraints**: Defines unique constraints on table columns.
  Ex. `@Table(uniqueConstraints = @UniqueConstraint(columnNames = {"email"}))`
  // email column value should be Unique

- **indexes**: Defines indexes for the table columns.
  Ex. `@Table(indexes = @Index(columnList = "name"))`

- **schema**: Defines the schema the Table belongs to.
  Ex. `@Table(schema = "HR")` - Maps to HR.EMPLOYEE_TABLE if name is also set.

2. Field Annotations

@Id

- Marks Field as Primary Key.

@GeneratedValue(strategy= _)

- Auto-generates the **primary key values** (IDENTITY, SEQUENCE, etc.).
  Ex. `@GeneratedValue(strategy=GenerationType.IDENTITY)`

➢ Attribute

- **strategy**: Defines how the value will be generated.
- **generator**: Used if you want a custom generator name

- **IDENTITY**: AUTO_INCREMENT (MySQL)

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private Long id;
```

- **SEQUENCE**: Uses @SequenceGenerator. Creates a sequence.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"employee_seq_gen")
@SequenceGenerator(name = "employee_seq_gen", sequenceName = "employee_seq",
allocationSize = 1)
private Long id;
```

- **TABLE**: Uses @TableGenerator. Creates a **separate ID table**.

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "emp_table_gen")
@TableGenerator(
      name = "emp_table_gen",
      table = "id_generator_table",
      pkColumnName = "gen_name",
      valueColumnName = "gen_value",
      pkColumnValue = "employee_id",
      allocationSize = 1 )
private Long id;
```

- **AUTO**: Chooses based on **dialect**. **Depends on DB.**

@Column(name="column_name")

- Used to specify details of the column that Maps to specific Field of an Entity.
  Ex. `@Column(name = "emp_name", nullable = false, length = 100, unique = true)`

➢ Attribute

- **nullable** : Default is true. | Column becomes NOT NULL. If (nullable = false)
- **unique** : Creates a unique constraint on the column.
- **length** : Default is 255. | Column type will be VARCHAR(100)
- **precision** : For BigDecimal type | precision = 10 | total digits - 10 | DECIMAL(10,2).
- **scale** : For BigDecimal | @Column(precision = 10, scale = 2) | 2 decimal place
- **insertable** : Default is true. | if false, column not be inserted new values.
- **updatable** : Default is true. | if false, column doesn't allow to update its values.
- **columnDefinition** : Defines a column as TEXT, even if the field is a String.

@Type

- To mention Type of data.

@Transient

- Used to mark a Property or Field in an Entity class as Transient.
- This means, hibernate can exclude that field from the persistence process and the corresponding column won't be created in DB table.

Syntax:

```
// exclude it from data persistence in the database.
@Transient
private int discountPrice = productPrice * 5 / 100;
```

@Temporal

`@Temporal(TemporalType.DATE)` : Stores only the date (Ex. DATE, TIME, TIMESTAMP).

@Embedded

- Used to 'Mark a Field' in a class as being an embeddable object.

@Embeddable

- Used to 'Mark a Class' as being embeddable.
- Meaning its properties can be included in another class as a value type.
- Used for Embedding a value object (a reusable component) into an entity.
- Using @Embeddable and @Embedded annotations, hibernate can automatically **persist the properties** of the **embeddable**(Address.java) **class** within the **containing** (Student.java) **class**.

```java
@Embeddable
public class OrderItemId implements Serializable {
    private Long orderId;
    private Long productId;

    // Constructors, Getters, Setters, equals(), and hashCode()
}
```

Syntax:

```
@Embeddable
public class Address { // fields}

@Entity
public class Employee {
        @Embedded
        private Address address;
}
```

@EmbeddedId

- Marks a **field** as a **composite primary key**.

```java
@Entity
public class OrderItem {

    @EmbeddedId
    private OrderItemId id;

    private int quantity;

    // Getters, Setters, and possibly @MapsId if referencing another entity
}
```

@Lob

- Indicates that the property should be stored in the database in the form of a large object type in the database. (large binary objects)
- Marks field as Large Object(LOB) for storing Large Data like Images/Text in DB.

Syntax:

```
@Lob
private byte[] photo;

@Lob
private byte[] reports;
```

@Version

- It provides the version number for a specific entity.
- Version Number is used to prevent concurrent modification to an entity.
- When an entity is being updated, the version number is also incremented.
- @Version annotation is used for optimistic locking.

Syntax:

```
Employee {                  // emp fields can be updated till version number only.
        @Version
        private int version;    // auto-increment
}
```

@DynamicInsert(true)

- Used to Exclude null columns in the INSERT statement of new objects.
- By default, hibernate includes all columns, even if some values are null.
- With @DynamicInsert, only non-null values are inserted.

Ex. `INSERT INTO employees (id, name, department) VALUES (1, 'John', NULL);` **//without**
`INSERT INTO employees (id, name) VALUES (1, 'John');` **//with**

@DynamicUpdate(true)

- Used to exclude unchanged columns in UPDATE statements of existing entity object.
- By default, hibernate updates all columns, even if only one field changed.
- With @DynamicUpdate, only modified fields are updated.

Ex. `UPDATE employees SET name = 'John', department = 'HR' WHERE id = 1;` **// without**
`UPDATE employees SET department = 'HR' WHERE id = 1;` **// with**

[dynamic-insert="true" | dynamic-update="true"]

How can you make a property be read from the database but not modified in anyway (make it immutable)?

1. **Using @Column Attributes**
   - Set the insertable = false and updatable = false flags on the property:

```
@Column(name = "created_at", insertable = false, updatable = false)
private LocalDateTime createdAt;
```

2. **For XML Mappings**
   - If you're using .hbm.xml mapping files:

```
<property name="createdAt" column="created_at" insert="false" update="false"/>
```

3. **Relationship Annotations**: Defines relationships between Tables.

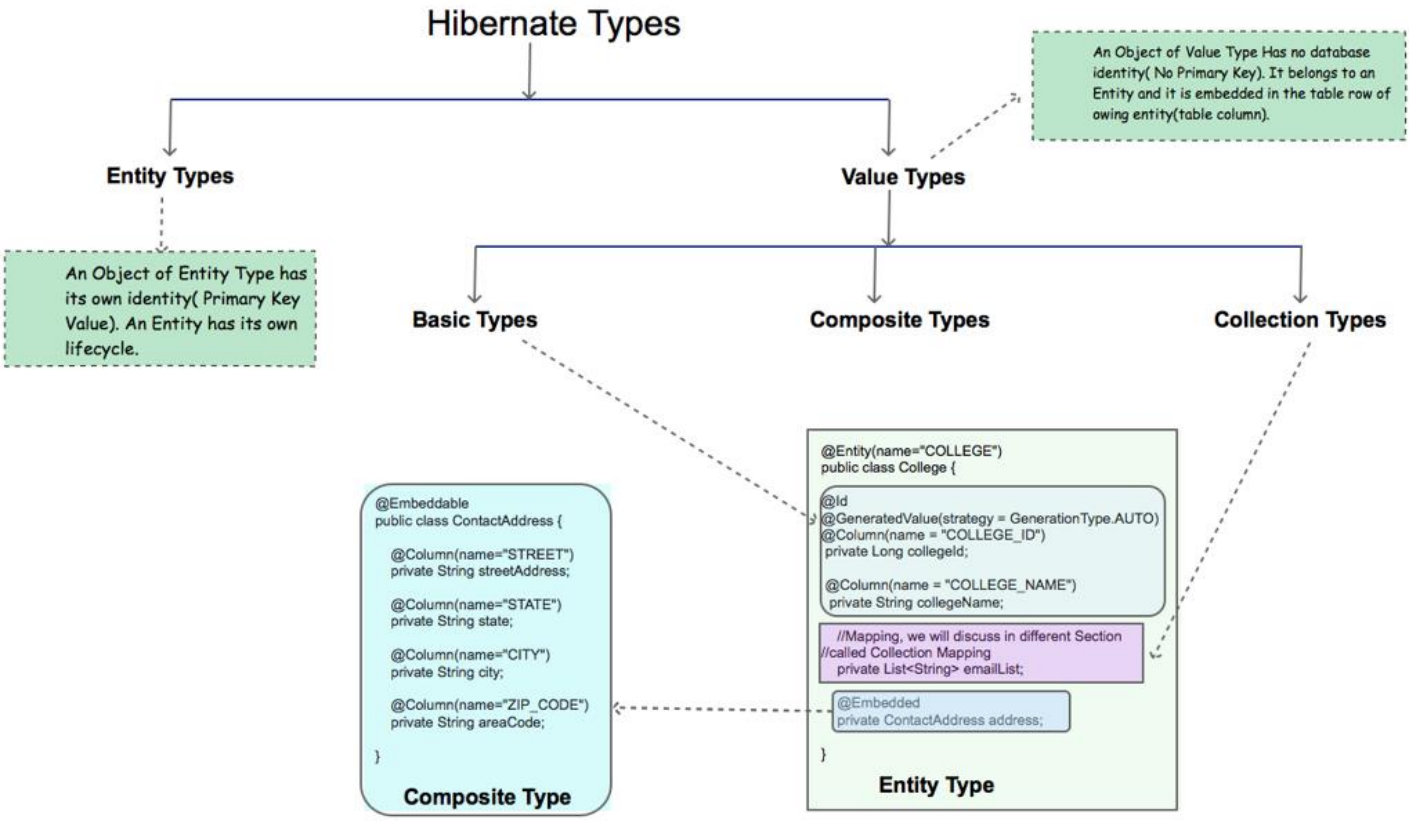
Why hibernate.cfg.xml Cannot Be Fully Replaced by Annotations?

- Annotations can **replace** most **mapping-related configurations** (like @Entity, @Table, @Column, etc.), but **not the core Hibernate configuration** like database connection, dialect, and other settings.
- `Alternative to hibernate.cfg.xml`

```
Configuration cfg = new Configuration();
cfg.setProperty("hibernate.connection.url", "jdbc:mysql://localhost:3306/mydb");
cfg.setProperty("hibernate.connection.username", "root");
cfg.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
cfg.addAnnotatedClass(Employee.class);
SessionFactory sessionFactory = cfg.buildSessionFactory();
```

Hibernate Mapping file: filename.hbm.xml

```
<?xml version="1.0"?> <!DOCTYPE hibernate-mapping PUBLIC
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
   <class name="sample.MyPersistanceClass" table="MyPersitaceTable">
      <id name="id" column="MyPerId">
         <generator class="increment"/>
      </id>
      <property name="text" column="Persistance_message"/>
      <many-to-one name="nxtPer" cascade="all" column="NxtPerId"/>
   </class>
</hibernate-mapping>
```


Hibernate Types
An Object of Value Type Has no database identity(No Primary Key). It belongs to an Entity and it is embedded in the table row of owing entity(table column).
Entity Types
Value Types
An Object of Entity Type has its own identity(Primary Key Value). An Entity has its own lifecycle.
Basic Types
Composite Types
Collection Types
@Embeddable
public class ContactAddress {
@Column(name="STREET")
private String streetAddress;
@Column(name="STATE")
private String state;
@Column(name="CITY")
private String city;
@Column(name="ZIP_CODE")
private String areaCode;
}
Composite Type
@Entity(name="COLLEGE")
public class College {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "COLLEGE_ID")
private Long collegeId;
@Column(name = "COLLEGE_NAME")
private String collegeName;
//Mapping, we will discuss in different Section
//called Collection Mapping
private List<String> emailList;
@Embedded
private ContactAddress address;
}
Entity Type

CRUD Operations

Save () vs persist ()

1. Serializable save(Object object)

- **Defined in: `org.hibernate.Session`**
- **Returns:** `The generated identifier (ID) of the inserted entity (Serializable)`
- **When to use:** If you want to get the **generated ID** immediately.
- **Transaction:** Can be called outside of a transaction. (**but not recommended**).
- **Duplicates:** Can insert **duplicate rows** if called multiple times with the **same object**.

```
Serializable id = session.save(employee);
```

2. void persist(Object object)

- **Defined in: `javax.persistence.EntityManager` (JPA)** (also in **`org.hibernate.Session`**)
- **Returns:** void (no return value)
- **When to use:** When you're using JPA and prefer cleaner code.
- **Transaction:** Must be called within an **active transaction**.
- **Duplicates: Doesn't** insert **duplicate rows**, **throws an exception** if the object is **already associated**.
- Method given by **JPA** and its **implementation provided in Hibernate.**
- Doesn't allow to work with 'Generators'

```
session.persist(employee);
```

Which One Should You Use?

- Use **save()**
  - If you need the **generated ID** immediately.
  - You're working in a **Hibernate-native** application.
- Use **persist()**
  - If you're following **JPA standards**.
  - You want **portable, provider-independent** code.

Selecting/Retrieving data from DB => **get() vs load()**

3. <T> T get(Class<T> entityClass, Serializable id)

- It performs **Eger Loading** By-default. whether to use or not, get() will load object.
- **Immediately hits the database** and loads the entity when called.
- If the entity **is present in session cache**, no DB call is made.
- If the entity **doesn't exist**, returns null.
- Method create **only one object** ie. Real Entity class object when it called.
- **Use When:**
  - You **need to confirm** that the object exists.
  - You want to **avoid lazy-loading exceptions**.

Ex.
```
Employee employee = session.get(Employee.class, 1);   // Immediate DB hit
System.out.println(employee.getName());               // Already fetched
```

4. <T> T load(Class<T> entityClass, Serializable id)

- It performs **Lazy Loading** of object By-default.
- Returns a **proxy object**, without immediately hitting the DB.
- The actual DB Query Triggered when the object is being used for the First Time.
- If **entity doesn't exist** in DB, throws **ObjectNotFoundException** on access.
- Method creates two objects for
  - Real Entity class
  - Proxy InMemmory Entity class. - **Proxy Class** will be the **child** of **real Entity Class**.

Ex.
```
Employee employee = session.load(Employee.class, 1);  // No DB hit yet
.....
..... // still DB Query not Triggered
System.out.println(employee.getName());               // Triggers a DB Query
```

Use When:

- You're **sure** the object exists and want to **delay DB access**.
- You want to **optimize performance** with lazy loading.

- Both methods take the entity class and the primary key of the object as arguments.
- Load() Can perform Eger loading also by adding lazy="false" in <class> tag of mapping file.

Hibernate Object Reattachment Methods

- When you modify a **detached entity** (an entity not associated with a Hibernate session), you need to **reattach** it before syncing with the database. Hibernate provides several methods for this:

5. update()

- It is used to Reattaches a **detached entity** to the session. **Detached → Persistent.**
- update() does not check whether changes were made, it simply marks the Entity as Persistent.
- Throws an **NonUniqueObjectException** if the same entity (with same ID) already associated with the session.

```
Employee emp = session1.get(Employee.class, 1L);
session1.close();  // emp is now detached
emp.setSalary(90000);  // change made in detached state
Session session2 = sessionFactory.openSession();
session2.update(emp);  // reattaches
session2.getTransaction().commit();
```

6. saveOrUpdate()

- Used to either **saves a new entity** or **updates an existing one** in the database. (detached/ transient).
- **Checks entity existence first** using a **SELECT** query.
- If Entity **doesn't exist** in the DB, saveOrUpdate() performs an **INSERT**.
- If Entity **already exists,** saveOrUpdate() performs an **UPDATE**.
- Throws an **NonUniqueObjectException** if the same entity (with same ID) already associated with the session.

```
session.saveOrUpdate(emp);  //INSERT or UPDATE based on select query result
```

7. merge()

- **Defined in: `org.hibernate.Session`** and **`javax.persistence.EntityManager`**
- Used to **reconnect a detached entity** to the current Hibernate session and **synchronize** it with the database.
- **Returns:** A **new managed instance** — the original object remains **detached**.
- Method generate **Select query** to verify **existence of Entity** first then it'll generate **Insert or Update** query based on Select Query Result. like saveOrUpdate().
- But merge () **doesn't throw NonUniqueObjectException** if the **same object** is already **in the session**.

- emp is detached (from session1)
- managedEmp is the **new** persistent instance in session2
- Changes in emp are copied to managedEmp

```
Session session1 = sessionFactory.openSession();
Employee emp = session1.get(Employee.class, 1);
session1.close(); // emp is now detached

emp.setName("Updated Name");

// New session
Session session2 = sessionFactory.openSession();
session2.beginTransaction();

Employee managedEmp = (Employee) session2.merge(emp);

session2.getTransaction().commit();
session2.close();
```

Use merge() when

1. Use **merge()** when you're working with **detached objects** (e.g., from another session or layer).
2. Always returns a **new managed object**, so don't expect your original object to become attached.
3. Let Hibernate to handle both insert and update operations automatically. (saveOrUpdate())

Avoid merge() when

1. Having an attached entity in the same session. (use update())

```
Session session1 = sessionFactory.openSession();
Employee emp = session1.get(Employee.class, 1L);
session1.close();

emp.setSalary(95000);  // changes in detached object

Session session2 = sessionFactory.openSession();
Employee managedEmp = (Employee) session2.merge(emp);  // new managed instance
session2.getTransaction().commit();
```

Best Practice

- Use merge() when dealing with detached objects to avoid session conflicts.
- Use update() if the entity is already attached to the session.
- Use saveOrUpdate() if you don't know whether the entity is new or existing.

➔ If level-1 cache of Session, having loaded the Entity class Object1 with Id and trying to save/update/saveOrUpdate/delete/persist the Entity class Object2 with same Id value then will get an Exception. ie. NonUniqueObjectException

Solution : use merge(entity object)

- **What methods must the persistent classes implement in Hibernate?**

Since Hibernate instantiates the persistent classes using Constructor.newInstance(), it requires a constructor with no arguments for every persistent class. And getter and setter methods for all the instance variables.

➔ **update () vs merge ()**

- Both are used to Synchronize Objects with the DB, but they behave differently when dealing with Attached and Detached Entities.

1. Purpose: Used **to attach a detached entity** back to the session and update it.
1. Purpose: Used to copy the state of a detached entity into a new managed entity inside the session.

2. Limitation: **Throws an Exception** if the Same Entity is already attached in the session.
2. Advantage: **Does not throw an exception** if the entity is already attached.

3. update() generates only Update query.
3. merge() generates first Select query then based on result, generates Insert/Update Query.

Feature	update()	merge()
API Source	Hibernate (org.hibernate.Session)	Hibernate + JPA (org.hibernate.Session, javax.persistence.EntityManager)
Returns	void	Returns a **new managed instance**
Works with Detached Object	Yes	Yes
Object Becomes Managed?	Yes — the **passed object** becomes managed	No — a **new managed copy** is returned
If Entity Already Exists in Session	Throws NonUniqueObjectException	Safe — merges data into the existing session entity
Can Insert New Row?	No — assumes object exists in DB	Yes — will insert if ID is not found in DB
Performance	Slightly faster (fewer internal checks)	Slightly heavier due to copying and checks
Best Use Case	When you're 100% sure the entity is **not in the session yet**	When you're working with **detached entities**, especially in complex flows
JPA-Compliant?	No	Yes

Best Practices for Fetching Data

- Optimize Fetch Strategy: Use lazy or eager loading based on requirements to minimize database queries.
- Pagination: Always use pagination for large datasets.
- Use Projections for Specific Fields: Avoid fetching unnecessary columns.
- Avoid N+1 Problems: Use joins or fetch joins for relationships when needed.

Hibernate Caching

- Hibernate caching stores entity data, queries, or collections in memory so Hibernate can **avoid hitting the database repeatedly** for the same data.
- To Enhance a performance of application by reducing DB access and reusing frequently accessed data.

Types of Caching in Hibernate

First-Level/session-level Cache

- The session-level cache is **enabled by default** and cannot be disabled.
- When an entity is loaded or updated for the first time in a session, it is stored in the session-level cache and cleared as Session closed.
- If trying to load the same entity within the same session, then same entity is served from Cache.
- Scope: Cache is per Session. Data cached in one session is not accessible in another session.

1st Time call

```
Employee emp1 = session.get(Employee.class, 34);
System.out.println(emp1.hashCode()+"\t"+emp1);        // 176191196
```

2nd Time Call

```
Employee emp2 = session.get(Employee.class, 34);
System.out.println(emp2.hashCode()+"\t"+emp2);        // 176191196

emp1 == emp2  : true
```

Second-Level/Factory-level Cache

- Hibernate provides several second-level cache providers that can be used to store cached data in a shared cache across multiple sessions.
- Scope: Cache is shared across multiple sessions and is configured at the **SessionFactory level**.
- Entity fetched in session A is cached and can be reused in session B without DB hit.
- When an entity is loaded or updated for the first time in a session, it is stored in the second-level cache.
- If trying to load the same entity within the Different session, then same entity will be served from Second level Cache.

→ **To configure the second-level cache in Hibernate,**

- Add dependency to pom.xml cache provider such as Ehcache, Infinispan, and Hazelcast.
- Configure Hibernate properties to enable the second-level cache and specify the caching provider.

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">
                org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
<property name="hibernate.cache.use_query_cache">true</property>
```

- Configure entity caching for specific entities in your Hibernate application.

```
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE, region="myEntityCache")
public class Employee { - }

<!-- https://mvnrepository.com/artifact/net.sf.ehcache/ehcache -->
<dependency>
<groupId>net.sf.ehcache</groupId>
<artifactId>ehcache</artifactId>
<version>2.10.6</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-ehcache -->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-ehcache</artifactId>
<version>5.6.15.Final</version>
</dependency>
```

Query Cache

- Definition: The query cache stores the results of HQL/JPQL or SQL queries. It is used alongside the Second-level cache.
- Scope: Cache is shared across sessions for specific queries. Must be explicitly enabled.
- No Need of @Cacheable.
- Enable query caching in hibernate.cfg.xml:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

- Mark the query as cacheable in your code:

```
query.setCacheable(true); // Enables query caching
```

Cache Concurrency Strategies

READ_ONLY: Used for data that does not change (e.g., reference tables).

READ_WRITE: Ensures strong consistency.

TRANSACTIONAL: Used for applications requiring ACID transactions.

ID Generators

Candidate Key Column

- The column having Unique value and can used to retrieve the record from DB, Known as Candidate Key Column
- Student Table with column = Adhar, DOB, Name, PAN, Course, Batch, Registration_No
  here, Adhar, PAN, Registration_No will be Candidate Key Column.

Natural Key Column

- The Candidate Key Column which is having outside business meaning.
- here, Adhar, PAN, VoterId, PassportNo will be Natural Key Column.

Surrogate Key Column

- The Candidate Key Column whose value doen't have business meaning and Generated dynamically by underlying Applications/Project/DB, Known as Surrogate Key Column.

  Ex. AutoIncrement(MySQL), Sequence(Oracle), Generators in Hiberanate etc.

While choosing any generator for your project make sure

- The value and length of generated Id value by Generator is compatible with Id property in Entity class.
- underlying DB must support generator.
- All Hibernate generator classes are implementing org.hibernate.id.IdentifierGenerator (I)

- Configure generators using <generator/> tag in entity_class.hbm.xml(Mapping) file inside <id/> tag.

```
<id name="id">
<generator class="assigned"/>
</id>
```

Note:

- Always take **Surrogate Key Column** as **Primary Key**.
- During SessionFactory creation, the object of generator class instantiated.

Generator

- Used to automatically generate and assign Unique Primary Key values for the @Id field.

1. Assigned

- The Assigned generator is configured by Default, if not configured any generator.
- Programmer or end user needs to manually set Id property value before inserting data to DB.

2. Auto

- Hibernate automatically chooses the best strategy based on the DB.
- Uses SEQUENCE (if supported) or IDENTITY.
- When you don't want to specify a generator strategy.

```
@GeneratedValue(strategy = GenerationType.AUTO)
```

```
<generator class="auto"/>
```

3. Increment

- Generates an incremental ID.
- Uses a formula (Max value + 1) to generate id property.
- generated value take priority over manually assigned property value.

```
@GeneratedValue(strategy = GenerationType.INCREMENT)
```

```
<generator class="INCREMENT"/>
```

4. Identity

- Uses auto-increment from the database. Uses a formula (Max value + 1) to generate id property.
- The IDENTITY generator automatically assigns primary keys using the database's AUTO_INCREMENT feature.
- No need for Hibernate to generate or manage the primary key.

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
<generator class="identity"></generator>
```

Not supported by Oracle (use SEQUENCE instead).

5. Sequence

- Uses a Database Sequence to generate unique IDs.
- More efficient than IDENTITY.
- Oracle, PostgreSQL (databases that support sequences).

```
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "emp_seq")
@SequenceGenerator(name = "emp_seq", sequenceName = "employee_seq", allocationSize = 1)
```

allocationSize = 1 → Generates one ID at a time (more database calls).
allocationSize = 10 → Fetches 10 IDs at once, reducing database calls.

- <sequence_name> start with <random_number> increment by <random_number>

```
<generator class="sequence">
        <param name="sequence">sequential_datasource</param>
</generator>
```

6. Hilo

- Uses a combination of high & low values to generate unique IDs.
- Large-scale applications needing better ID generation performance.
- Uses a formula = hi_value * (max_lo +1)

```
@GeneratedValue(generator = "hilo_gen")
@GenericGenerator(name = "hilo_gen", strategy = "hilo")
```

```
<generator class="hilo">
        <param name="table">document</param>
        <param name="column">column</param>
        <param name="max_lo">13210</param>
</generator>
```

7. UUID

- A 128-bit value used to uniquely identify information.
- Commonly shown as: 550e8400-e29b-41d4-a716-446655440000
- Hibernate provides a built-in UUID generator using @GeneratedValue and @GenericGenerator.
- Use Java's UUID class (Recommended for Hibernate 5.0+)

  - Hibernate will auto-generate UUIDs using java.util.UUID.randomUUID().
  - Requires a DB column of type UUID or VARCHAR(36) (depends on dialect).

```
@Id
    @GeneratedValue
    private UUID id;
```

-OR-

```
  @Id
    @GeneratedValue(generator = "uuid-gen")
    @GenericGenerator(name = "uuid-gen", strategy = "uuid2")
    private UUID id;
```

```
    <generator class="uuid"> </generator>
```

Ex: **"id": "3f0a9932-842a-4d0c-8d1c-f8a5b2cf1e66"**

- Make sure the column in the DB is compatible with the UUID format you're using.

Database	UUID Column Type
PostgreSQL	UUID
MySQL	CHAR(36) or BINARY(16)
Oracle	RAW(16) or VARCHAR2(36)

Composite Mapping

What is Composite Mapping in Hibernate?

- It's a technique in Hibernate where a **primary key** is made up of **multiple columns** i.e., a composite key.
- Especially useful when you have tables with multiple columns acting as a primary key in your database.

```
@Embeddable
public class OrderDetailId implements Serializable {
        private Long orderId;
        private Long productId;
        // equals() and hashCode() are mandatory!
}

@Entity
@Table(name = "orders")
public class OrderDetail {

        @EmbeddedId
        private OrderDetailId id;       // orderId + productId

        private int quantity;
        private double price;
        // getters and setters
}
```

- **@Embeddable** marks the **composite key class**.
- **@EmbeddedId** marks the **field** inside the **entity**.

Hibernate Inheritance

- Relational database tables **don't naturally support inheritance**.
- In Hibernate, can map **a class inheritance** hierarchy to **database tables.**

➢ If you have a class hierarchy like this:

```
@Entity
@Inheritance
public class Vehicle {
    @Id
    private Long id;
    private String brand;
}
```

```
@Entity
public class Car extends Vehicle {
    private int seats;
}

@Entity
public class Bike extends Vehicle {
    private boolean hasCarrier;
}
```

What are the different approaches to represent an inheritance hierarchy?

- Table per **Class-Hierarchy.**
- Table per **Concrete class.**
- Table per **Subclass**.

Inheritance Strategies

1. **Single Table Strategy** (Table Per Hierarchy - **SINGLE_TABLE**)

- **Default inheritance strategy** in JPA/Hibernate. One table for **entire class hierarchy.**
- Stores all entities (parent + child classes) in **a single database table**.
- Uses a **discriminator column** to distinguish between different subclass types.
- Saves space (only one table) but leads to **nullable columns** for fields not used by all **subclasses**.

Ex. **Vehicle** table store objects of all 3 classes **Vehicle**, **Car** and **Bike.**

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "vehicle_type", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue("Vehicle")
public class Vehicle {
   @Id
   private Long id;
   private String brand;
}

@DiscriminatorValue("Car")
public class Car extends Vehicle {
   private int seats;
}

@DiscriminatorValue("Bike")
public class Bike extends Vehicle {
   private boolean hasCarrier;
}
```

id	brand	seats	hasCarrier	vehicle_type
1	Honda	5	null	Car
2	Yamaha	null	true	Bike

- Many **null** columns for **unused fields**

2. Table Per Class Strategy (Table per Concrete class **- TABLE_PER_CLASS)**

- Each **concrete subclass** has its **own database table** with Its own fields and Fields inherited from the superclass
- The superclass has **no separate table** of its own.
- All inherited fields (from the superclass) are **duplicated** in **each subclass table**.
- There is **no need for a discriminator column**.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
   @Id
   private Long id;
   private String brand;
}
@Entity
public class Car extends Vehicle {
   private int seats;
}
@Entity
public class Bike extends Vehicle {
   private boolean hasCarrier;
}
```

Car Table:

id	brand	seats
1	Honda	5

Bike Table:

id	brand	hasCarrier
2	Yamaha	true

Drawbacks:

- **Data redundancy**: Same columns (e.g., brand, id) repeated in all subclass tables.
- **Slower polymorphic queries**: Uses UNION across multiple tables.
- Harder to enforce **foreign key relationships** with parent.

3. Joined Table Strategy (JOINED)

- Each class in the inheritance hierarchy (both **superclass and subclasses**) has **its own separate table** in the database.
- Subclass tables contain **only subclass-specific fields** and a **foreign key (same as primary key)** that references the **superclass table**.
- Hibernate uses **SQL joins** to assemble the full entity data.
- Uses **@PrimaryKeyJoinColumn** on **subclass** to indicate **join column** (usually the primary key).
- Ensures **normalized schema** (no duplication of fields like in TABLE_PER_CLASS).
- Requires **joins** at runtime to fetch complete object (slower than SINGLE_TABLE for reads).

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Vehicle {
  @Id
  private Long id;
  private String brand;
}

@Entity
@PrimaryKeyJoinColumn(name = "id")
public class Car extends Vehicle {
  private int seats;
}

@Entity
@PrimaryKeyJoinColumn(name = "id")
public class Bike extends Vehicle {
  private boolean hasCarrier;
}
```

Vehicle Table:

id	brand
1	Honda
2	Yamaha

Car Table:

id	seats
1	5

Bike Table:

id	hasCarrier
2	true

- **Use SINGLE_TABLE** for performance & simple hierarchies.
- **Use JOINED** for normalized schema & better DB design.
- **Use TABLE_PER_CLASS** when subclasses are fully independent.

Drawbacks:

- **Requires joins** for every query (can impact performance).

- Slightly **slower** than SINGLE_TABLE for retrieval.
- **Slightly more complex** database schema.

Hibernate Relationships/Association

- In a relational DB, Tables can have **relationships** (**associations**) between them.
- In Hibernate, we map these relationships using association mappings between entities(classes).

Types of Relationships

Relationship Type	Description	Example
@OneToOne	**One entity relates to one other entity**	**A person has one passport**
@OneToMany	**One entity relates to many entities**	**A department has many employees**
@ManyToOne	**Many entities relate to one entity**	**Many employees belong to one department**
@ManyToMany	**Many entities relate to many others**	**A student enrolled in many courses**

@OneToOne

- Defines a **one-to-one relationship** between **two entities**.
  Ex: User ↔ Address
  - A User has **one** Address, and
  - Each Address belongs to **exactly one** User.

```
@Entity
public class User {
        @Id
        private int userId;

        @OneToOne
        @JoinColumn(name = "add_id", referencedColumnName = "address_id")
        private Address address;  // FK is in User table
}

@Entity
public class Address {
   @Id
   private int address_id;

   @OneToOne(mappedBy = "address") // Inverse side
   private User user;
}
```

- **@JoinColumn:** Specifies that **User** holds the **foreign key (add_id)** pointing to **Address**.
- **mappedBy = "address":** Declares **Address** as the **inverse side**.
- This means the **foreign key column (add_id) exists only in the User table**, not in Address.

@OneToMany + @ManyToOne (Bidirectional)

@OneToMany

- Defines a one-to-many relationship (e.g., one parent can have many children).
- ***one-to-many* mapping means that one row in a table is mapped to multiple rows in another table.**

@ManyToOne

- Defines a many-to-one relationship (e.g., many orders belong to one customer).
- It is used when **multiple instances of one entity are associated with a single instance of another entity.**
- Consider two entities: **Employee and Department**. Each employee belongs to a single department, but a department can have multiple employees.
- where multiple employees (many) can be associated with a single department (one).

```java
@Entity
@Table(name = "departments")
public class Department {

@Id
private Long id;

private String name;

@OneToMany(mappedBy = "department")
private List<Employee> employees;
//…
```

```java
@Entity
@Table(name = "employees")
public class Employee {

@Id
private Long id;

private String name;

@ManyToOne
@JoinColumn(name = "department_id")
private Department department;

// Constructors, getters, and setters
}
```

- **mappedBy = "department"** tells Hibernate that the department field in **Employee owns the relationship. Employee table have foreign key as 'department_id'**
- The **@JoinColumn** annotation is used to specify the **foreign key column** in the employees table that references the departments table.

```java
Department department = new Department();
department.setId(1L);
department.setName("HR");

Employee employee1 = new Employee();
employee1.setId(1L);
employee1.setName("John Doe");

employee1.setName(department);



Employee employee2 = new Employee();
employee1.setId(2L);
employee1.setName("Will Smith");
```

```
employee1.setName(department);
```

```
@OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Task> empTask = new ArrayList<>();

@ManyToOne
@JoinColumn(name = "EMP_ID", referencedColumnName = "empId") // FK in Task table
private Employee employee;
```

@ManyToMany

- Defines a **many-to-many** relationship (e.g., students and courses).
- Each side can have **multiple associations** with the other.
- **one entity** has **multiple instances** (collections) of the **other entity** and vice versa.
- *Employees* and *Projects* can have *many-to-many* relationships.
- In this case, an *Employee* may work on multiple *Projects*, and a *Project* may have multiple *Employees*.

```
@Entity
public class Employee {
        @Id
        private Long empId;

        @ManyToMany(cascade = CascadeType.ALL)
        @JoinTable(
          name = "EMP_PROJ_TBL",
          joinColumns = @JoinColumn(name = "EMP_ID"),
          inverseJoinColumns = @JoinColumn(name = "PROJECT_ID")
        )
        private List<Project> projects = new ArrayList<>();
}

@Entity
public class Project {
  @Id
  private Long projectId;

  @ManyToMany(mappedBy = "projects")
  private List<Employee> employees = new ArrayList<>();
}
```

Attribute of : @OneToOne, @OneToMany, @ManyToOne, @ManyToMany

- **cascade = CascadeType.ALL**
  - Used to specify cascading operations (e.g., PERSIST, MERGE, etc.) for related entities.
    - **PERSIST** - save new child if parent is saved
    - **MERGE** - update child if parent is updated
    - **REMOVE** - delete child if parent is deleted
    - **REFRESH** - refresh child if parent is refreshed
    - **ALL** - All the above.

- **fetch = FetchType.LAZY**
  - **LAZY** : Load the related entity only when accessed (proxy).
    - default for @OneToMany, @ManyToMany
  - **EAGER** : Load the related entity immediately.
    - default for @ManyToOne, @OneToOne

- **mappedBy = "employee"**
  - Defines the which table can have Foreign Key Column. for Bi-directional mapping.
    - eg. User(Target) has one Address(Dependent). both class have @OneToOne
  - The side with @JoinColumn (User) is the owning side.
  - The side with mappedBy (Address) is the inverse side

- **orphanRemoval = true**
  - If a User is removed, its Address can be removed automatically if this is true.

@JoinColumn(name = "column_name")

- Defines the Foreign Key column for @OneToOne, @ManyToOne or @ManyToOne a relationship.

Attribute : name, nullable, unique

Eg. @JoinColumn(name = "department_id") // Foreign Key : department_id

@JoinTable(name = "table_name")

- Defines a Join table for @ManyToMany relationships.

Attribute : name, joinColumns, inverseJoinColumns

```
Eg.
@JoinTable( name = "student_course",
      joinColumns = @JoinColumn(name = "student_id"),
      inverseJoinColumns = @JoinColumn(name = "course_id")
)
```

Fetching Technique

- In Hibernate, two common strategies for **fetching Associated entities** are "**Lazy Loading**" and "**Eager Loading**":

```
@Entity
public class Department {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String name;

 @OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
 private List<Employee> employees;
}

@Entity
public class Employee {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String name;

 @ManyToOne(fetch = FetchType.LAZY)
 private Department department;
}
```

- **LAZY Fetching (FetchType.LAZY)**
  - With **Lazy Loading**, **Associated entities** are **not fetched** from the database **immediately** when the **parent entity** is **retrieved**.
  - Instead, Associated entities are loaded from the database only **when** they are **accessed for the first time.**
  - Data is fetched **on-demand** (only when it is accessed for the first time)
  - Hibernate uses **proxy** (copies) **objects** to load the **Associated** data when needed.

Ex:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "department")
private List<Employee> employees;
```

Explanation: The list of employees will not be fetched from the database until you explicitly access it.

```
Department department = session.get(Department.class, 1);
// Employees are NOT fetched yet
SOUT(department.getName());     // select query executed
SOUT(department.getEmployees()); //Employees fetched only when accessed
```

- **EAGER Fetching (FetchType.EAGER)**
  - **Eager Loading** fetches **Associated** entities from the database along with the **Parent** entity during the **initial retrieval.**
  - Data is fetched **immediately** along with the **parent entity**, even if it's not accessed.
  - This leads to additional **JOIN queries or SELECT queries** at the time of loading the **parent entity.**

  Ex:

  ```
  @OneToMany(fetch = FetchType.EAGER, mappedBy = "department")
  private List<Employee> employees;
  ```

  # Explanation: The list of employees will be fetched as soon as the department entity is loaded.

- **Use LAZY Fetching:**
  - When related data is large or not always required.
  - Example: A User entity with a list of Orders. You don't always need to fetch all orders when retrieving a user.
- **Use EAGER Fetching:**
  - When related data is small and is always required.
  - Example: A Book entity with an Author. The author is almost always needed when fetching the book.

Association Type	Default Fetch Type
@ - ToOne	EAGER
@ - ToMany	LAZY

What is Dirty Checking in Hibernate?

Hibernate automatically detects **changes** made to persistent entities (objects) during a Session, and the **automatic generate SQL UPDATE statements** to save those changes to the database.

- When an entity is **fetched from the database**, Hibernate stores an **original snapshot** of the entity.
- During the same **open session**, if you **modify any fields** of the entity, Hibernate compares the current state with the original snapshot.
- On calling session.flush() or transaction.commit(), Hibernate checks if any fields have changed (i.e., the entity is "dirty").
- Then automatically **executes SQL UPDATE** only **for the modified columns.**

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Employee emp = session.get(Employee.class, 1); // Persistent
emp.setName("John Updated");                   // Modify field

// No session.update() called!

tx.commit(); // Dirty Checking triggers: Hibernate auto-generates UPDATE
session.close();
```

UPDATE Employee SET name = 'John Updated' WHERE id = 1;

You didn't explicitly call session.update(), but Hibernate still updates the DB because of **dirty checking**.

- Dirty checking happens only for **persistent** entities (those managed by Hibernate inside a session).
- You don’t need to explicitly call `session.update()` for it to work.
- You can disable dirty checking with **@DynamicUpdate** or manual flushing.

Pagination

- **Pagination** is the process of dividing a large dataset into smaller subsets (pages) so that only a **limited number of records** are loaded and displayed at a time.
- Involves dividing the data into fixed size "**chunks**" or "**pages**," and then displaying only one page at a time.

Advantage

- By limiting the amount of data that is loaded at once, pagination can improve the performance of web pages, reduce the load time and improve the user experience.
- The pagination implemented using the setFirstResult() and setMaxResults() of the Query(I).

Hibernate provides multiple ways to implement pagination using:

1. **HQL (Hibernate Query Language)**

```
String hql = "FROM User";
Query query = session.createQuery(hql);

// Set pagination parameters
query.setFirstResult(0);      // Offset (start index, 0-based)
query.setMaxResults(10);      // Page size (number of records per page)

List<User> users = query.list();
```

2. **Criteria API**
   - Criteria is **deprecated** in Hibernate 5+ in favor of **JPA CriteriaBuilder**.

```
Criteria criteria = session.createCriteria(User.class);
criteria.setFirstResult(0);      // Start from record 0
criteria.setMaxResults(10);      // Limit to 10 records

List<User> users = criteria.list();
```

3. **CriteriaBuilder (JPA 2.0)**

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> root = cq.from(User.class);
cq.select(root);

TypedQuery<User> query = entityManager.createQuery(cq);
query.setFirstResult(0);     // Offset
query.setMaxResults(10);     // Limit

List<User> result = query.getResultList();
```

4. **Native SQL Queries**

```
Query query = session.createNativeQuery("SELECT * FROM users");
        query.setFirstResult(0);
        query.setMaxResults(10);

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

1. setFirstResult(int firstResult)

- To set the index of the first record to retrieve in a query result set.
- index 0 - n | records 1 - n (table record)

2. setMaxResults(int maxResults)

- To set the maximum number of records to retrieve in a query result set.
- The specified number of records will be retrieved from the result set, starting from the index set by the setFirstResult().

Syntax

```
String hql = "FROM User";
Query query = session.createQuery(hql);   // HQL
query.setFirstResult(0);                  // Offset (start position)
query.setMaxResults(maxResults);          // Page size (number of records per page)
List<User> users = query.list();
```

In Hibernate, pagination directly using HQL, Criteria, CriteriaBuilder, or Native Queries.

➢ **EntityManager**

- *EntityManager* is part of the Java Persistence API.
- it implements the programming interfaces and lifecycle rules defined by the JPA.
- EntityManagerFactory provides instances of EntityManager for connecting to same database. All the instances are configured to use the same setting as defined by the default implementation.
- JPA EntityManager is used to access a database in a particular application. It is used to manage persistent entity instances, to find entities by their primary key identity, and to query over all entities.
- For ***each HTTP request 1 EntityManager*** will be created.

SQL DIALECT

- A **Dialect** in Hibernate is a configuration component that **tells Hibernate how to generate static or dynamic SQL Queries** for a specific **database type** (MySQL, Oracle, PostgreSQL, etc.)
- When Hibernate builds the **SessionFactory**, it uses the **Dialect** to generate SQL for:
    - Table creation (CREATE TABLE)
    - Insert/Update/Delete
    - Sequences/auto-increment
    - Joins, limits, pagination, etc.
- All dialects are hibernate api that are extending **org.hibernate.dialect.Dialect**

  <property name="hibernate.dialect">
  org.hibernate.dialect.**MySQLDialect**</property>

  Hibernate: **`insert into product (p_name,p_quantity,p_price,p_id)`**
  `values (?,?,?,?)`
- Whenever calling **session.save(), session.update(), or tx.commit(),** This query will be created or ganerated.
- It generates **DDL (schema) queries** at startup, but **DML (INSERT/UPDATE)** queries are generated **at runtime** when you perform actions.

How to generate Dynamic insert or update SQL query?

- In **Mapping** File add attribute in <class> tag
  dynamic-insert="true" **and** dynamic-update="true"

<class name="com.hibernate.entities.Product" table="product" dynamic-insert="true">

Criteria Value

- Any **single row object** is fetched by providing an **identity value** that is called as **Criteria Value.**

➢ To perform bulk row operations like **insert, delete** then should go for **HQL, Criteria API.**

hbm2ddl.auto

- This feature makes hibernate framework to generate DB table Dynamically based on given Info.
- To use this, It is recommeneded to to specify Type, Length, Not Null, Unique, etc. attributes.

Values:

1. create

- Always Re-Create DB table/schema at each startup based on hibernate Mappping File information.
- If DB table already exist then it'll delete table and create New fresh table.
- All data is lost after every restart.

  Syntax <property name="hbm2ddl.auto">create</property>

2. update (best)

- It creates DB Table if the DB table is not already available.
- If DB table is already available, then it won't create any table.
- Existing data remains intact.
- Best for development when schema changes frequently.

  Syntax <property name="hbm2ddl.auto">update</property>

3. validate (default)

- Hibernate checks if tables/columns exist but does not modify the schema.
- If there is a mismatch as configured in Mapping file, hibernate throws SchemaManagementException.

  Syntax <property name="hbm2ddl.auto">validate</property>

4. create-drop

- Drops and recreates the schema at the beginning.
- Deletes the schema when the SessionFactory is closed.

  Syntax <property name="hbm2ddl.auto">create-drop</property>

Can We Modify Entity Object Without Calling update ()?
How to synchronize Entity Class object to DB Table record?

- **Yes**, but only when the entity is in **the persistent state.**
- Hibernate **automatically tracks changes** to persistent entities and synchronizes them with the database **during flush() or commit()**.

Synchronizing Entity to Database:

1. **When the Entity is Persistent**

- If Entity is still associated with active Session(persistent state), any Modification to the Entity is Automatically Detected by Hibernate, and when committed transaction, changes will be saved.
  Ex :
  Employee emp = session.get(Employee.class, 1); // Entity in Persistent state
  emp.setSalary(75000); // Modify entity
  tx.commit(); // Hibernate automatically updates the record
  session.close();

Why?

- Hibernate uses a mechanism called **Dirty Checking**. It compares the **current entity state** with the **snapshot** it took when it was **first loaded**. If it **detects changes**, it generates an **UPDATE SQL automatically** on **commit**.

2. When is the Entity Detached (After Session is Closed)?

- Detached means **the Session is closed**, and the entity is no longer tracked.
- Modifying the entity now will **not reflect in the database** unless you re-attach it.

Ex:

```
Session session1 = sessionFactory.openSession();
Employee emp = session1.get(Employee.class, 1);
session1.close();        // emp is now in Detached state
emp.setSalary(80000);    // Change will NOT be saved yet
```

- Now, to save this modified detached object: **Use merge ()**

```
// Open a new session and merge the changes
Session session2 = sessionFactory.openSession();
Transaction tx = session2.beginTransaction();

session2.merge(emp); // Returns a new persistent copy with merged changes

tx.commit();
session2.close();
```

Why?

- merge() copies changes from the detached object into a **new managed instance**.
- It is **safe even if another Session already has the same entity ID loaded**.
- Unlike update(), it does not throw **NonUniqueObjectException**.

3. In Spring Boot using @Transactional (With JPA/Hibernate)

- When you use **Spring Data JPA + Hibernate + @Transactional**, Spring manages: **Session lifecycle, Transaction**
- Spring's @**Transactional** automatically calls **flush**() and **commit**() behind the scenes, ensuring Hibernate **syncs changes to the DB.**
- Using Spring Boot with Hibernate, don't need update() because Spring manages transactions for you.
- **Why:** The @Transactional annotation ensures that Hibernate automatically updates the entity at the end of the transaction.

```
@Transactional
public void updateSalary(Long id) {
    Employee emp = employeeRepository.findById(id).orElseThrow();
    emp.setSalary(90000);        // Automatically tracked
    // No need to call save() or update()
} // On method exit, transaction commits & changes persist
```

Is it possible to declare mappings for multiple classes in one mapping file?

- Yes, by using multiple **<class>** elements. But, the **recommended** practice is to use **one mapping file per persisten class**.

```
<hibernate-mapping>
    <class name="com.example.User" table="users">
        <id name="id" column="user_id" />
        <property name="name" column="username" />
    </class>

    <class name="com.example.Product" table="products">
        <id name="id" column="product_id" />
        <property name="name" column="product_name" />
    </class>
</hibernate-mapping>
```

How are the individual properties mapped to different table columns?

- In Hibernate, individual properties of an entity are mapped to columns in a database table using either XML or annotations.
- Each field is mapped to a column using @Column(name = "column_name").
- Map the Java fields **name** and **email** to the **username** and **email_address** columns in the **users** table.

```
<class name="com.example.User" table="users">
    <id name="id" column="user_id">
        <generator class="identity"/>
    </id>

    <property name="name" column="username"/>
    <property name="email" column="email_address"/>
</class>
```

What are derived properties?

- **Derived properties** in Hibernate are **properties that are not stored directly in the database**, but are **calculated or derived** from other columns or expressions—often using SQL formulas or transient logic.

```
private String firstName;
private String lastName;

@Formula("concat(first_name, ' ', last_name)")
private String fullName;

// `fullName` is derived, not stored
```

1. **Formula-Based Derived Property**

- Hibernate will generate a SELECT that includes the computed column:

`SELECT id,first_name,last_name, concat(first_name, ' ', last_name) AS fullName FROM employee;`

2. **Transient Derived Property**

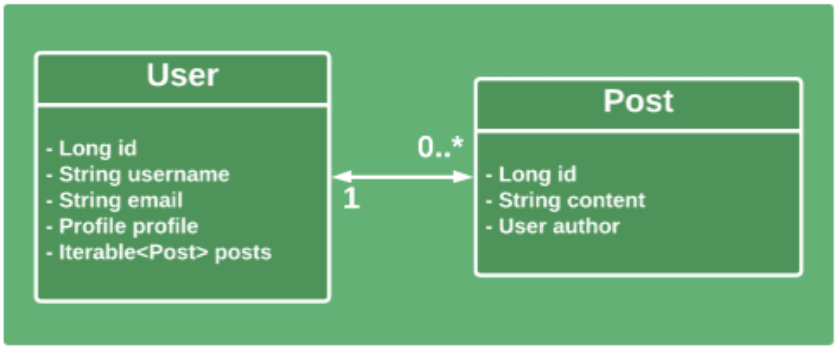
```
@Transient
public double getTotalPrice() {
    return price + tax;
}

// `getTotalPrice()` won't be persisted
```

❖ N + 1 Select Problem

1. What is the N+1 Select Problem?

- The **N+1** problem in Hibernate refers to a **performance issue** that arises when **fetching** a **collection of entities** (**N parent records**) and their associated child entities, causing **N Additional Database Queries** to be executed - one for each parent entity.
- This leads to increased **database load**, slower **response times**, and **poor** application **performance**, especially with **large datasets**.
- Slows down **web pages or APIs** displaying parent-child data.

Ex: If there are N **parent** entities (**User**) and each has a one-to-many relationship with **child** entities (**Posts**), Hibernate may issue **N+1 Queries** to fetch **all parent entities (Users)** and then later fetch **each set of** related **child entities (Posts) individually**.




```
@Entity
@Data
public class User {
    @Id
    private Long id;
    private String username;
    private String email;

    @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy =
"author")
    protected List<Post> posts;
}

@Entity
@Data
public class Post {
 @Id
 @GeneratedValue (strategy = GenerationType.IDENTITY)
 private Long id;
 private String name;

 @ManyToOne(fetch = FetchType.LAZY)
 private User author;
}
```

- The **Fetch type** of relationships is assumed as **default if not mentioned explicitly**.
- All ***to-one*** relationships have **eager fetch** and ***to-many*** have **lazy fetch**.
- This problem typically appears in **@OneToMany** or **@ManyToMany** relationships where the **child (Posts) collection is lazily loaded**, especially **when iterating over them.**

Scenario:

- let's consider the relationship between **User** and **Post Entities**.
- If we have **5 Users** and we iterate over them to retrieve their **Posts** using **Lazy Loading**, Hibernate will execute **1 query** to fetch **all Users** and then, **for each User**, it will execute an **additional query** to retrieve **its Posts**.
- This results in a total of **6 queries** (**1 for Users + 5 for Posts**), even though a single query could have fetched all the required data efficiently.

```
List<User> users = userRepository.findAll();
for (User user : users) {
  List<Post> posts = user.getPosts();              // Triggers lazy load per user
}
```

Hibernate will perform:

- One query to load all users:
  - SELECT * FROM user;
- One query per user to fetch their posts:
  - SELECT * FROM post WHERE author_id = 1;
  - SELECT * FROM post WHERE author_id = 2;
  - …

Total queries = 1 (for users) + N (for posts) = N+1 queries

Why Does the N+1 Problem occur?

- The **N+1** problem arises in Hibernate, when the **Lazy Loading** is implemented for **Associations** between **Entities**.
- When **Lazy Loading** is used, related Entities (**Posts**) are fetched from the Database only when they are **accessed for the First time**.

  Ex: if there are **N** entities (**User**) in the main collection, Hibernate will execute **N+1** Queries to fetch the related entities (**Posts**), this can lead overhead and poor performance.

Possible Solutions for N+1 Problem:

- **JPQL with JOIN FETCH (Java Persistence Query Language)**
  - Use a join fetch query to fetch all related entities in a single query, avoiding the N+1 query problem.

```
@Query ("SELECT u FROM User u JOIN FETCH u.posts")
List<User> findAllUsersWithPosts();
```

Single query with a join: `SELECT * FROM user u JOIN post p ON u.id = p.author_id;`

Why use JPQL?

- Works across any supported database (DB-agnostic).
- Operates on **JPA entity models** (not raw DB schemas).
- Portable and type-safe compared to native SQL.

Basic Syntax: `SELECT e FROM EntityName e WHERE e.property = :value`

Ex:

```
@Query("SELECT u FROM User u JOIN FETCH u.posts")
List<User> findAllUsersWithPosts();
```

- **Entity Graph**

```
@EntityGraph (attributePaths = "posts")
@Query ("SELECT u FROM User u")
List<User> findAllWithEntityGraph();
```

- **DTO Projections**

```
@Query ("SELECT new com.example.UserPostDTO(u.name, p.title) FROM User u JOIN
u.posts p")
List<UserPostDTO> findUserPostDetails();
```

Properties of **hibernate.cfg.xml :**

 # JDBC Properties

connection.driver_class : Represents the JDBC driver class.
connection.url : Represents the JDBC URL.
connection.username : database username.
connection.password : database password.
connection.pool_size : Represents the Max no of connections available in the connection pool.

 # Datasource Properties

connection.datasource : Represents datasource JNDI name, used by Hibernate for DB properties.
jndi.url : Represents the URL of the JNDI provider. <optional>
jndi.class : Represents the class of the JNDI InitialContextFactory.<optional>

 # Configuration Properties

dialect : Represents the type of database used in hibernate to generate SQL statements
for a particular relational database.
sql_show : Display the executed SQL Query on console.
format_sql : Display the executed SQL Query in formated way on console.
default_catalog : Qualifies unqualified table names with the given catalog in generated SQL.
default_schema : Qualifies unqualified table names with the given schema in generated SQL.

 # Cache Properties

cache.provider_class : Represents the classname of a custom CacheProvider.
cache.use_query_cache : It is used to enable the query cache.

 # Other

hbm2ddl.auto : Automatically generates schema in the database with the creation of SessionFactory.
connection.provider_class : Represents Custom ConnectionProvider which provides JDBC connections to Hibernate.

Using Hibernate: Without Annotation

1. Entity class
2. HBM file (Mapping)
3. CFG file (Config)
4. Hibernate jar
5. main class(App.java)

EXAMPLE

- Configuration file "**hibernate.cfg.xml**" => src/main/resources

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3306/myhiber</property>
<property name="connection.username">root</property>
<property name="connection.password">pwd</property>
<property name="connection.pool_size">1</property>

<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
<property name="hbm2ddl.auto">create</property> // new table created | update
<property name="show_sql">true</property>       // show query on console
<property name="format_sql">true</property>
<property name="current_session_context_class">thread</property>
<property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider
</property>

// incases using .xml
<mapping resource="Project.hbm.xml"/>
(or)
// incase using annotation
<mapping class ="com.hibernate.Student"/>

</session-factory>
</hibernate-configuration>
```

- Mapping File "**class_name.hbm.xml**"

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
        <class name="com.hibernate.entity.Employee">
        <id name="id">
                <generator class="assigned"></generator>
        </id>
        <property name="name"></property>
        <property name="age"></property>
        <property name="salary"></property>
        </class>
</hibernate-mapping>
```

- Entity class (Employee.java)

```
@Data
@Entity
public class Employee {
@Id
@GeneratedValue(Strategy="GenerationType.AUTO")
        private int id;

        private String name;
        private int age;
        private long salary;
}
```

- **App.java**

return config | return factory | return session | return trx
configure() -> buildSessionFactory() -> getCurrentSession() -> beginTransaction()

```
App {
     Configuration config=new
     Configuration().configure("hibernate.cfg.xml");
     SessionFactory factory = config.buildSessionFactory();

     // get Session
     Session session = factory.getCurrentSession(); or openSession();
     Transaction tx = session.beginTransaction();

     // create Student object and pass data.
     Student st = new Student();
     session.save(st); // persist(emp)
     tx.commit();      // Actual query will be executed and generated here

     sout(factory);
     session.close();
}
```

HQL (Hibernate Query Language)

- Object-oriented query language provided by Hibernate that operates on entity objects rather than directly on database tables.

Writing and Executing HQL Queries

=> Queries are executed using the Query or TypedQuery interface.

Example 1: Retrieving All Entities

```
String hql = "FROM Employee"; // Entity name, not table name

Query<Employee> query = session.createQuery(hql, Employee.class);
List<Employee> employees = query.list();

for (Employee emp : employees) {
System.out.println(emp.getName());
}
```

Example 2: Using WHERE Clause

```
String hql = "FROM Employee WHERE department = :dept";

Query<Employee> query = session.createQuery(hql, Employee.class);

query.setParameter("dept", "IT");

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

for (Employee emp : employees) {
System.out.println(emp.getName());
}
```

HQL CRUD Operations

createQuery(hql) | setParameter("id", 1) | executeUpdate() | list()

=> Insert is performed by creating and saving an object:

```
Employee employee = new Employee();
employee.setName("John");
employee.setDepartment("HR");

session.save(employee);
```

=> Update: Modify existing records using executeUpdate().

```
String hql = "UPDATE Employee SET salary = :salary WHERE id = :id";
Query query = session.createQuery(hql);
query.setParameter("salary", 50000);
query.setParameter("id", 1);

int result = query.executeUpdate();
System.out.println(result + " rows affected.");
```

=> Delete: Remove records from the database.

```
String hql = "DELETE FROM Employee WHERE id = :id";
Query query = session.createQuery(hql);
query.setParameter("id", 1);
int result = query.executeUpdate();
System.out.println(result + " rows deleted.");
```

Using Joins in HQL

HQL supports joins to fetch associated data between entities.

Inner Join :

```
String hql = "SELECT e.name, d.name FROM Employee e INNER JOIN
e.department d";
Query<Object[]> query = session.createQuery(hql);
List<Object[]> results = query.list();

for (Object[] row : results) {
System.out.println("Employee: " + row[0] + ", Department: " + row[1]);
}

e.department = apply inner-join on employee class Dept column.

String hql = "FROM Department d JOIN FETCH d.employees WHERE d.id =
:id";
Query<Department> query = session.createQuery(hql, Department.class);
query.setParameter("id", 1);
Department dept = query.uniqueResult();
System.out.println("Department: " + dept.getName());
```

Pagination in HQL

HQL supports pagination using setFirstResult() and setMaxResults().

```
String hql = "FROM Employee";
Query<Employee> query = session.createQuery(hql, Employee.class);

query.setFirstResult(0); // Start index
query.setMaxResults(10); // Number of records to fetch

List<Employee> employees = query.list();
for (Employee emp : employees) {
System.out.println(emp.getName());
}
```

list() Method

- Fetches the object proxy(copy of object) and only hits the database when the object is accessed.
- It is called on a Query or Criteria object and returns a List.

```
Session session = factory.openSession();
List<Employee> employees = session.createQuery("from Employee",
Employee.class).list();
for (Employee emp : employees) {
System.out.println(emp.getName());
}
session.close();

uniqueResult() Method

Session session = factory.openSession();
Employee employee = session.createQuery("from Employee where id = :id",
Employee.class)
.setParameter("id", 1)
.uniqueResult();
System.out.println(employee.getName());
session.close();
```

- Fetches a single unique result.
- Throws NonUniqueResultException if the query returns more than one result.

==========

Spring JDBC Framework

- Spring JDBC is a Model class.
- Getter and setters are mandatory.
- The parameterized constructor will be helpful.
- There is no specific annotation is required. The only thing is we should have equal attributes match with the DB table and each attribute should have a getter and setter.
- Data Access Layer is specified with the interface and its implementation.

JdbcTemplate :

- This will take care all about common logic since creating connection with DB to closing the resources.

DataSource(I)
|
DriverManagerDataSource(C)

DataSource - Provides the connection to the database.

JdbcTemplate(C)

RowMapper and ResultSetExtractor

- Used to map rows from the database to Java objects.

Successfull Connection

1. Add Dependencies
   spring-core
   spring-context
   spring-jdbc
   mysql-connector-java

2. Configure JdbcTemplate

```
XML-Configuration - config.xml
create connection using DriverManagerDataSource
driverClassName, url, Username, Password

inject to JdbcTemplate
p:dataSource-ref=""
```

3. Java-Configuration

```
getDataSource() {
     DriverManagerDataSource ds = new DriverManagerDataSource();
     ds.setDriverClassName("com.mysql.jdbc.Driver");
     ds.setUrl("jdbc:mysql://localhost:3306/springjdbc");
     ds.setUsername("root");
     ds.setPassword("Mysqsk45@123");
     return DS;
}
public JdbcTemplate getJdbcTemplate() {
     JdbcTemplate jdbcTemplate = new JdbcTemplate();
     jdbcTemplate.setDataSource(getDataSource());
}
```

4. perform CRUD operations

```
Insert         - jdbcTemplate.update()
Update         - jdbcTemplate.update()
Delete         - jdbcTemplate.update()
getStudent     - jdbcTemplate.queryForObject(query, rowMapper,
studentId)
getAllStudent - jdbcTemplate.query(selectQuery, new RowMapperImpl())
```

==========