Object-Relational Mapping (ORM)

ORM is a technique that allows developers to convert data between incompatible type systems in object-oriented programming languages. ORM tools like Hibernate map Java objects to database tables and manage the relationships between these objects and the database.

### ORM Tools

Object-Relational Mapping (ORM) tools in Java are used to map Java objects to database tables and vice versa. There are number of ORM tools available, these tools are designed to simplify database interactions in Java applications by reducing the need to write extensive boilerplate code and by providing features that make working with databases more efficient and maintainable.

Here are some of the most popular ORM tools for Java:

1. Hibernate:

Hibernate is the most widely used ORM tool for Java. It provides a framework for mapping an object-oriented domain model to a relational database. It also implements the Java Persistence API (JPA), making it a popular choice for Java developers.

- **Automatic table generation**
- **HQL (Hibernate Query Language) for database operations**
- **Support for complex associations (e.g., one-to-one, one-to-many, many-to-one, many-to-many)**
- **Caching mechanisms to improve performance**
- **Support for lazy loading and fetching strategies**

2. EclipseLink:

EclipseLink is the reference implementation of the JPA specification and provides advanced features beyond the standard JPA. It was originally developed by Oracle and is now maintained by the Eclipse Foundation.

- **Advanced caching**
- **Native SQL queries**
- **Database platform independence**
- **Support for various data sources, including relational, XML, and NoSQL databases**

3. MyBatis:

MyBatis is a semi-ORM tool that focuses on mapping SQL statements to Java methods. Unlike Hibernate or JPA, it does not map entire objects to tables, allowing more control over SQL execution.

- **Flexible SQL mapping**
- **XML or annotation-based configuration**
- **Support for stored procedures**
- **Dynamic SQL generation**
- **Support for complex mappings like nested objects and lists**

4. Apache Cayenne:

Cayenne is a full-featured ORM framework that offers a model-driven approach, which means you can design your database schema and generate Java code accordingly.

- **Visual modeler for designing database schemas**
- **Support for inheritance, relationships, and multi-table joins**
- **Built-in connection pooling**
- **Web-friendly, supporting various web frameworks**
- **Flexible and extensible architecture**

5. OpenJPA:

OpenJPA is another implementation of the JPA specification and is maintained by the Apache Software Foundation. It offers a range of advanced features for enterprise-grade applications.

- **Optimistic and pessimistic locking**
- **Advanced caching mechanisms**
- **Support for complex queries**
- **Integration with various data sources and environments**

6. JOOQ (Java Object Oriented Querying):

JOOQ is an innovative ORM tool that turns SQL queries into type-safe Java code. It's not a full ORM tool but focuses on making SQL a first-class citizen in Java applications.

- **Type-safe SQL query construction**
- **Supports most SQL dialects**
- **Excellent support for complex queries**
- **Integration with various database systems**
- **Code generation from database schemas**

Jakarta Persistence (formerly Java Persistence API - JPA)

The Jakarta Persistence API (JPA) is a specification within the Jakarta EE (formerly Java EE) platform that provides a framework for managing relational data in Java applications. It defines a set of interfaces and standards for object-relational mapping (ORM), which allows developers to interact with databases using Java objects, rather than writing raw SQL queries.

JPA is a specification and not a tool itself. However, it defines a set of rules and guidelines for ORM tools. Many tools, including Hibernate, EclipseLink, and others, implement the JPA specification.

- **Standardized ORM for Java**
- **Annotations and XML descriptors for mapping**
- **Query language (JPQL)**
- **Entity lifecycle management**
- **Transaction management**

### Key Features of JPA:

1. Object-Relational Mapping (ORM): JPA allows you to map Java classes to database tables and Java objects to rows in those tables. This mapping is typically done through annotations or XML configuration.

2. Entity Management: In JPA, a database table is represented by an entity class. Each instance of an entity class corresponds to a row in the table. JPA provides mechanisms to perform CRUD (Create, Read, Update, Delete) operations on these entities.

3. EntityManager: The `EntityManager` is the primary interface used to interact with the persistence context. It handles operations such as persisting entities, removing entities, querying the database, and managing transactions.

4. Query Language: JPA includes JPQL (Java Persistence Query Language), a query language similar to SQL but designed to work with JPA entities. JPQL is database-agnostic, meaning it works across different database systems.

5. Transactions: JPA provides transaction management features, allowing developers to manage database transactions declaratively or programmatically.

6. Cascading and Relationships: JPA supports relationships between entities, such as one-to-one, one-to-many, many-to-one, and many-to-many. It also supports cascading operations, where actions on one entity can cascade to related entities.

7. Lifecycle Callbacks: JPA allows developers to define methods in entity classes that will be called automatically at certain points in the entity's lifecycle, such as before an entity is persisted or after an entity is updated.

8. Criteria API: In addition to JPQL, JPA offers a Criteria API for building type-safe, dynamic queries programmatically, which is useful when you need to construct queries dynamically at runtime.

### Benefits of Using JPA:

- Simplified Data Access: JPA abstracts the underlying database interactions, allowing developers to work with Java objects instead of raw SQL.

- Portability: JPA is a standard specification, so applications developed using JPA can be ported across different Jakarta EE-compliant application servers and database systems.

- Integration with Jakarta EE: JPA integrates seamlessly with other Jakarta EE technologies like Jakarta CDI (Contexts and Dependency Injection) and Jakarta Transaction.

JPA is often used with Hibernate, which is a popular implementation of the JPA specification, offering additional features beyond the standard.

Hibernate

Hibernate is an object-relational mapping (ORM) framework for Java that provides a framework for mapping an object-oriented domain model to a relational database. It simplifies the development of Java applications to interact with the database by allowing developers to work with objects rather than SQL queries.

https://hibernate.org/orm/documentation/6.5/

https://docs.jboss.org/hibernate/orm/6.5/userguide/html_single/Hibernate_User_Guide.html

https://docs.jboss.org/hibernate/orm/6.5/introduction/html_single/Hibernate_Introduction.html

https://docs.jboss.org/hibernate/orm/6.5/querylanguage/html_single/Hibernate_Query_Language.html

https://docs.jboss.org/hibernate/orm/6.2/javadocs/

### What is Hibernate?

Hibernate is an open-source object-relational mapping (ORM) framework for Java applications. It provides a framework for mapping an object-oriented domain model to a relational database, simplifying the development process by allowing developers to interact with the database using Java objects instead of SQL queries.

Hibernate was developed by Gavin King in 2001 to provide a better persistence framework than EJB (Enterprise JavaBeans) and other ORM tools available at the time. It abstracts the complexities of database interactions, making it easier for developers to work with data in a more natural, object-oriented way.

### Benefits of Hibernate for Application Development

- **Simplifies Development: Hibernate reduces the need for extensive JDBC code, making database interactions easier.**
- **Database Independence: Hibernate abstracts database-specific code, allowing for easy migration between different databases.**
- **Automatic Table Creation: Hibernate can automatically create and update database tables based on the Java class structure.**
- **Caching: Hibernate provides caching mechanisms to improve application performance.**
- **Lazy Loading: Hibernate supports lazy loading, which means that associated data is loaded on demand, improving efficiency.**
- **HQL and Criteria Queries: Hibernate offers powerful query capabilities through HQL (Hibernate Query Language) and Criteria queries.**

### Comparing Hibernate with EJB

- **Ease of Use: Hibernate is generally easier to use and configure compared to EJB.**
- **Lightweight: Hibernate is considered lightweight, whereas EJB is heavier and more complex.**
- **Flexibility: Hibernate offers more flexibility in terms of configuration and customization.**
- **Performance: Hibernate can be more performant due to its caching and lazy loading features.**
- **Transaction Management: EJB has built-in support for distributed transactions, whereas Hibernate typically relies on third-party transaction management systems.**

### Hibernate Framework

The Hibernate framework consists of several key components:

1. **Core: Provides the core functionality of Hibernate.**

2. **Annotations: Allows configuration using Java annotations.**

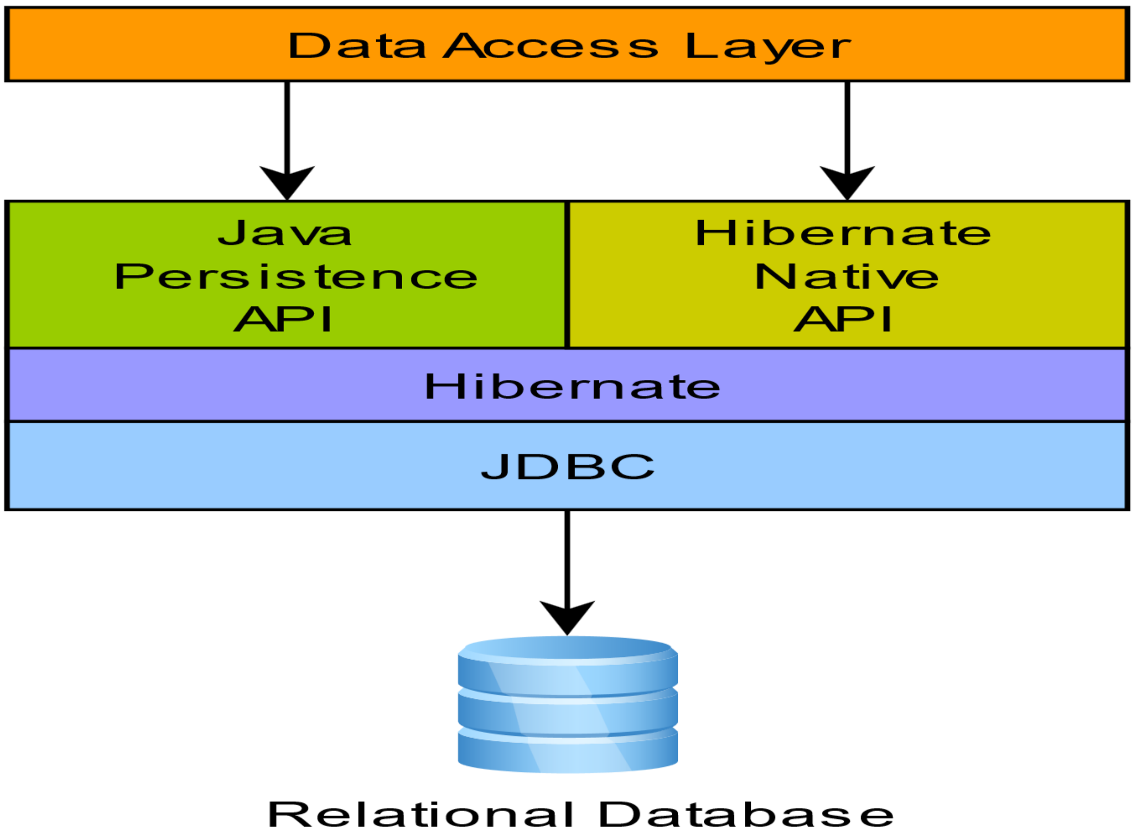
3. **EntityManager: Implements the JPA (Java Persistence API) standard.**

4. **Validator: Provides a framework for validating entities.**

5. **Search: Integrates full-text search capabilities into Hibernate.**

### Hibernate Architecture

The Hibernate architecture includes several layers:



The Hibernate architecture is structured around the following core components:

1. Configuration Object: Configures Hibernate and provides a way to load Hibernate settings from configuration files (usually `hibernate.cfg.xml`).

2. SessionFactory: A factory for `Session` objects. It is created once during application initialization and is an immutable, thread-safe object.

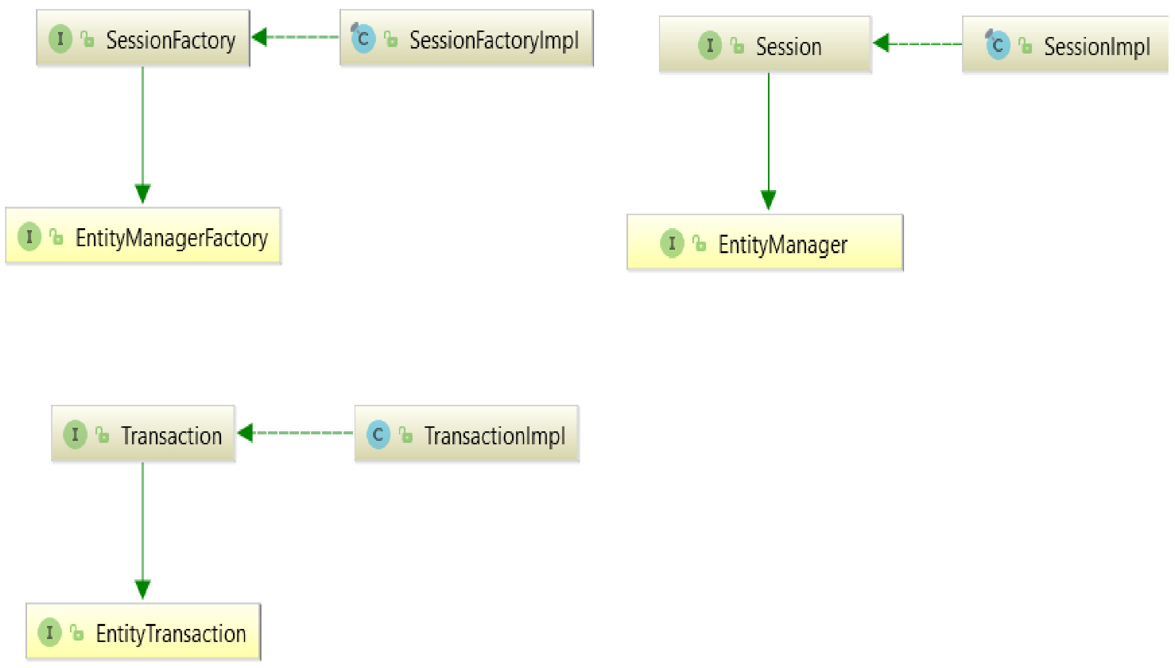
3. Session: A single-threaded, short-lived object representing a conversation between the application and the database. It provides methods to create, read, and delete operations for instances of mapped entity classes.

4. Transaction: Manages transaction boundaries. It abstracts application code from the underlying transaction management system.

5. Query: Represents a Hibernate query object, used to execute queries against the database.

6. Criteria: Provides a simplified API for retrieving entities by composing criteria objects.

As a Jakarta Persistence provider, Hibernate implements the Java Persistence API specifications and the association between Jakarta Persistence interfaces and Hibernate specific implementations can be visualized in the following diagram



### Compatibility

Hibernate 6.5.2.Final requires the following dependencies (among others):

Table 1. Compatible versions of dependencies

	Version
Java Runtime	11, 17 or 21
Jakarta Persistence	3.1.0
JDBC (bundled with the Java Runtime)	4.2

If you get Hibernate from Maven Central, it is recommended to import Hibernate Platform as part of your dependency management to keep all its artifact versions aligned.

### BOM stands for "Bill of Materials".

A Bill of Materials (BOM) is a special kind of POM (Project Object Model) file that defines a set of dependencies and their versions, which can be used across multiple projects. It's essentially a centralized repository of dependency versions that can be easily managed and shared.

A BOM typically contains:

1. Dependency versions: A list of dependencies with their specified versions.
2. Dependency management: A way to manage dependencies and their versions across multiple projects.

By using a BOM, you can:

1. Simplify dependency management: No need to specify versions for each dependency in every project.
2. Ensure consistency: Use the same versions across multiple projects.
3. Easily update dependencies: Update versions in one place (the BOM) and it will be reflected in all projects.

### How to integrate Hibernate and other related dependencies in a Maven project

Using the Jakarta EE Platform BOM and hibernate BOM is a good practice when working with multiple Jakarta EE APIs, ensuring that your project remains consistent and easy to maintain.

Check POM File :

https://github.com/anirudhagaikwad/Servlet_SpringBoot/blob/master/Practicals/HibernateJPA_JSP_APP/pom.xml

### Generator Class

The generator class in Hibernate is used to generate unique identifiers for the database records. Common generator strategies include:

Hibernate provides several generator classes to automatically generate primary key values for entities. These generators are part of the `@GeneratedValue` annotation, which is used in conjunction with the `@Id` annotation to define the primary key generation strategy. Here are some of the key generator classes provided by Hibernate:

### 1. AUTO

This is the default generation strategy, and it allows Hibernate to choose the appropriate generation strategy based on the underlying database.

```
@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;
```

### 2. IDENTITY

This strategy relies on the identity column of the database to generate the primary key. It is mainly used when the database supports auto-increment columns (e.g., MySQL, PostgreSQL).

```
@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;
```

### 3. SEQUENCE

This strategy uses a database sequence to generate the primary key. It is typically used with databases that support sequences (e.g., Oracle, PostgreSQL).

```
@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_gen")

@SequenceGenerator(name = "seq_gen", sequenceName = "my_sequence", allocationSize = 1)

private Long id;
```

### 4. TABLE

This strategy uses a separate database table to generate the primary key values. It is more flexible but less efficient than the `IDENTITY` or `SEQUENCE` strategies.

```
@Id

@GeneratedValue(strategy = GenerationType.TABLE, generator = "table_gen")

@TableGenerator(name = "table_gen", table = "id_gen", pkColumnName = "gen_name", valueColumnName = "gen_value", allocationSize = 1)

private Long id;
```

### 5. UUID

This strategy generates UUID values as primary keys. This is useful for systems where global uniqueness is required, and it does not rely on the database to generate the key.

```
@Id

@GeneratedValue(generator = "UUID")

@GenericGenerator(name = "UUID", strategy = "org.hibernate.id.UUIDGenerator")

private UUID id;
```

### 6. Custom Generators

Hibernate allows you to create custom primary key generators by implementing the `IdentifierGenerator` interface. This is useful if the provided generators do not meet your needs.

```
@Id

@GeneratedValue(generator = "custom-gen")

@GenericGenerator(name = "custom-gen", strategy = "com.example.CustomIdGenerator")

private Long id;
```

### 7. Hilo

Uses the hi/lo algorithm to generate identifiers.

### Dialects

In Hibernate, dialects refer to a set of classes that define the specific SQL syntax and behavior for a particular relational database management system (RDBMS). Different databases have variations in SQL syntax, functions, and features. Hibernate abstracts these differences by using dialect classes to generate the appropriate SQL for the database you're using.

Hibernate supports various SQL dialects to accommodate different types of databases. Each dialect allows Hibernate to generate optimized SQL for the specific database. Some common dialects include:

- **org.hibernate.dialect.MySQLDialect**
- **org.hibernate.dialect.OracleDialect**
- **org.hibernate.dialect.PostgreSQLDialect**

### Mapping

Mapping is the process of linking a class to a database table. It can be done using:

- XML Mapping Files: Define mappings in XML files.
- Annotations: Use JPA annotations directly in the Java classes.

https://github.com/anirudhagaikwad/Servlet_SpringBoot/blob/master/Practicals/HibernateNativeJSP_APP/src/main/resources/hibernate.cfg.xml

### Annotations

Hibernate uses JPA annotations for mapping entities to database tables:

- `@Entity`: Declares a class as an entity.

- `@Table`: Specifies the table name.

- `@Id`: Declares the primary key.

- `@GeneratedValue`: Specifies the strategy for generating primary key values.

- `@Column`: Specifies the column name.

- `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`: Defines relationships between entities.

In Hibernate and JPA, the annotations `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany` are used to define relationships between entities in a relational database. These annotations help map the object-oriented relationships in your Java code to the relational model of the database.

1. @OneToOne

A `@OneToOne` relationship indicates that one entity is associated with exactly one instance of another entity. It is a one-to-one association, meaning each row in the primary table corresponds to a single row in the associated table.

Use Case: When two entities share a one-to-one relationship, such as a user and their profile.

Example: A `Person` entity may have a one-to-one relationship with a `Passport` entity, where each person has exactly one passport, and each passport is associated with exactly one person.

```
@Entity

public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "passport_id", referencedColumnName = "id")
    private Passport passport;
}
@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToOne(mappedBy = "passport")
    private Person person;
}
```

2. @OneToMany

A `@OneToMany` relationship indicates that one entity is associated with multiple instances of another entity. It is a one-to-many association, meaning each row in the primary table can be associated with multiple rows in the related table.

Use Case: When an entity has a collection of another entity, such as a department containing multiple employees.

Example: A `Department` entity may have a one-to-many relationship with an `Employee` entity, where each department can have multiple employees, but each employee belongs to only one department.

```
@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Employee> employees;
}
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

3. @ManyToOne

A `@ManyToOne` relationship indicates that multiple instances of an entity are associated with a single instance of another entity. It is the inverse of `@OneToMany`. This association is often used to represent a foreign key relationship.

Use Case: When an entity belongs to another entity, such as an employee belonging to a department.

Example: Continuing the `Department` and `Employee` example, each employee belongs to one department, but a department can have many employees.

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
}
```

4. @ManyToMany

A `@ManyToMany` relationship indicates that multiple instances of one entity are associated with multiple instances of another entity. This relationship requires a join table to hold the associations between the two entities.

Use Case: When entities have a many-to-many relationship, such as students enrolling in courses.

Example: A `Student` entity may have a many-to-many relationship with a `Course` entity, where each student can enroll in multiple courses, and each course can have multiple students.

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}

@Entity
public class Course {
    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @ManyToMany(mappedBy = "courses")

    private List<Student> students;

}
```

### Summary:

- @OneToOne: Each entity instance is associated with one and only one instance of another entity.
- @OneToMany: An entity instance is associated with multiple instances of another entity.
- @ManyToOne: Multiple instances of an entity are associated with one instance of another entity.
- @ManyToMany: Multiple instances of an entity are associated with multiple instances of another entity, requiring a join table.

### CascadeTypes

In Hibernate and JPA, CascadeTypes define the operations that should be propagated (or cascaded) from a parent entity to its related child entities. Cascading is essential in maintaining the consistency of the database when operations are performed on entities that have relationships with other entities. By specifying a cascade type, you control how related entities are treated when the parent entity undergoes operations like persist, merge, remove, etc.

### Cascade Types in JPA and Hibernate:

1. CascadeType.PERSIST:

When the parent entity is persisted (i.e., saved to the database), the related entities are also persisted automatically.

Example: If you persist a `Department` entity, and it has a collection of `Employee` entities, all the `Employee` entities will be automatically persisted as well.

```
@OneToMany(cascade = CascadeType.PERSIST)
```

```java
    private List<Employee> employees;
```

2. CascadeType.MERGE:

When the parent entity is merged (i.e., updated or synchronized with the database), the related entities are also merged automatically.

Example: If you merge a `Department` entity, the changes made to its related `Employee` entities will also be merged.

```java
    @OneToMany(cascade = CascadeType.MERGE)

    private List<Employee> employees;
```

3. CascadeType.REMOVE:

When the parent entity is removed (i.e., deleted from the database), the related entities are also removed automatically.

Example: If you remove a `Department` entity, all its related `Employee` entities will be deleted from the database as well.

```java
    @OneToMany(cascade = CascadeType.REMOVE)

    private List<Employee> employees;
```

4. CascadeType.REFRESH:

When the parent entity is refreshed (i.e., reloaded from the database), the related entities are also refreshed automatically.

Example: If you refresh a `Department` entity, all its related `Employee` entities will also be refreshed, discarding any unsaved changes.

```java
    @OneToMany(cascade = CascadeType.REFRESH)

    private List<Employee> employees;
```

5. CascadeType.DETACH:

When the parent entity is detached from the persistence context (i.e., the entity manager is no longer tracking it), the related entities are also detached.

Example: If you detach a `Department` entity, all its related `Employee` entities will also be detached, meaning they are no longer managed by the entity manager.

```java
    @OneToMany(cascade = CascadeType.DETACH)

    private List<Employee> employees;
```

6. CascadeType.ALL:

This is a shorthand for applying all of the above cascade types (`PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`).

Example: Using `CascadeType.ALL` on a relationship will ensure that all operations performed on the parent entity will be cascaded to the child entities.

```
@OneToMany(cascade = CascadeType.ALL)

private List<Employee> employees;
```

### Why Cascade Types Matter:

- Consistency: Cascade types help maintain consistency in the database by ensuring that related entities are automatically persisted, updated, or deleted when the parent entity undergoes these operations.
- Convenience: They reduce the amount of code you need to write by allowing you to handle relationships as part of the parent entity's operations.
- Control: Different cascade types provide fine-grained control over which operations should affect related entities.

By using cascade types appropriately, you can simplify your code and ensure that your application maintains the correct state in the database.

### Transaction Management

Transaction management in Hibernate is crucial for ensuring that your database operations are executed in a consistent, reliable, and atomic manner. Transactions allow you to group a set of operations (such as inserting, updating, or deleting records) into a single, indivisible unit of work. If any operation within the transaction fails, the entire transaction can be rolled back to maintain data integrity.

### Key Concepts in Hibernate Transaction Management

1. Transaction:

A transaction is a sequence of operations performed as a single logical unit of work. Transactions in Hibernate typically start with a `beginTransaction` call and end with a `commit` or `rollback` operation.

2. ACID Properties:

Transactions are designed to adhere to the ACID properties:

- Atomicity: Ensures that all operations within a transaction are completed successfully or none at all.
- Consistency: Guarantees that the database remains in a consistent state before and after the transaction.
- Isolation: Ensures that transactions are executed in isolation from each other, preventing concurrent transactions from interfering.
- Durability: Once a transaction is committed, the changes are permanent, even in the event of a system failure.

3. Transaction Boundaries:

Typically, a transaction begins when you call `beginTransaction()` and ends with either `commit()` (if the transaction is successful) or `rollback()` (if an error occurs).

### How to Manage Transactions in Hibernate

Hibernate provides several ways to manage transactions, either programmatically or declaratively. Here are the common approaches:

### 1. Programmatic Transaction Management

In programmatic transaction management, you explicitly control the transaction boundaries using the `Transaction` API provided by Hibernate.

```
Session session = null;

Transaction transaction = null;

try {

    session = sessionFactory.openSession();

    transaction = session.beginTransaction();

    // Perform database operations

    Employee employee = new Employee();

    employee.setName("John Doe");
```

```
        session.save(employee);

        // Commit the transaction

        transaction.commit();

    } catch (Exception e) {

        if (transaction != null) {

            transaction.rollback(); // Rollback in case of an error

        }

        e.printStackTrace();

    } finally {

        if (session != null) {

            session.close(); // Always close the session

        }

    }
```

### 2. Declarative Transaction Management

Declarative transaction management typically relies on using Spring Framework's transaction management features with Hibernate. This approach allows you to manage transactions declaratively, usually through annotations or configuration files, without manually controlling transaction boundaries in your code.

### 3. Transaction Isolation Levels

Transaction isolation levels determine how isolated a transaction is from the effects of other concurrent transactions. Hibernate supports different isolation levels:

- READ_UNCOMMITTED: Allows dirty reads (reading uncommitted changes from other transactions).
- READ_COMMITTED: Prevents dirty reads, but non-repeatable reads and phantom reads can occur.
- REPEATABLE_READ: Prevents dirty and non-repeatable reads, but phantom reads can occur.
- SERIALIZABLE: The highest level, preventing dirty, non-repeatable, and phantom reads by serializing access to data.

The isolation level can be set in Hibernate configuration:

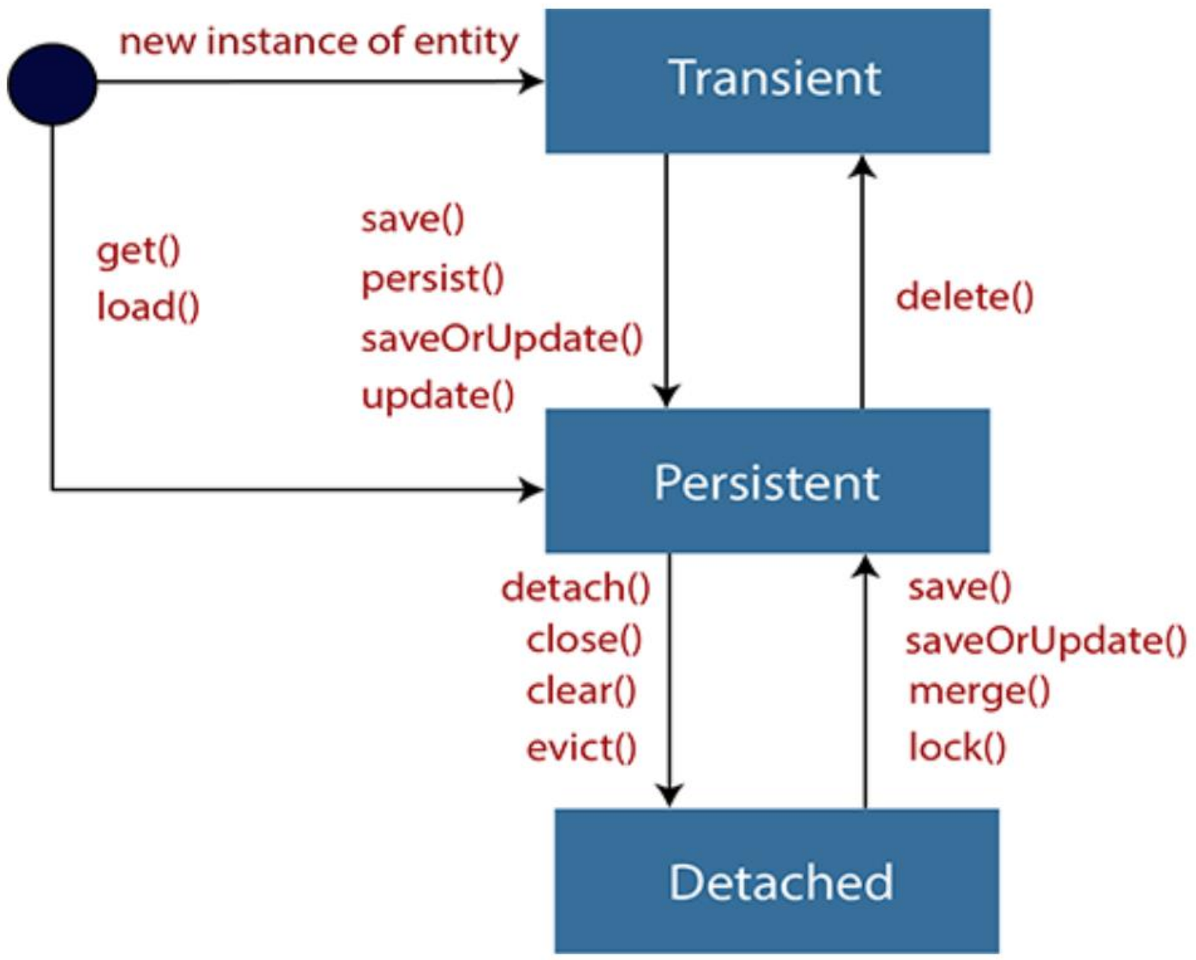
```
<property name="hibernate.connection.isolation">2</property>
```

Where `2` stands for `READ_COMMITTED`, `1` for `READ_UNCOMMITTED`, `4` for `SERIALIZABLE`, and `3` for `REPEATABLE_READ`.

### 4. Handling Transactions in Multi-Threaded Environments

In multi-threaded environments, each thread should manage its own transaction and session. Hibernate sessions are not thread-safe, so sharing a session or transaction across multiple threads can lead to concurrency issues.

Hibernate itself does not have a lifecycle in the same way that entities or Java EE components (like servlets) do. Instead, Hibernate deals with the lifecycle of entities and sessions. Here's a breakdown of these lifecycles:



### 1. Entity Lifecycle

Hibernate manages the lifecycle of entities through different states as they are processed by the persistence context. The primary states are:

1. Transient:

An entity is in the transient state if it has just been instantiated and is not yet associated with a Hibernate `Session`. It has no representation in the database and does not have an identifier assigned by the database.

```
Employee employee = new Employee(); // Transient state
```

2. Persistent:

An entity becomes persistent when it is associated with an active Hibernate `Session` and is represented in the database. Changes to the entity are automatically synchronized with the database.

```
Session session = sessionFactory.openSession();

session.beginTransaction();

Employee employee = new Employee();

session.save(employee); // Persistent state

session.getTransaction().commit();

session.close();
```

3. Detached:

An entity is in the detached state when it was previously associated with a `Session` but is no longer associated with any `Session`. It has been removed from the persistence context but still exists in the application.

```
Employee employee = session.get(Employee.class, id); // Detached state after session close
```

4. Removed:

An entity is in the removed state when it has been marked for deletion but the changes have not yet been synchronized with the database. Once the transaction is committed, the entity is deleted from the database.

```
session.delete(employee); // Removed state

session.getTransaction().commit();
```

### 2. Session Lifecycle

The `Session` is a fundamental part of Hibernate's architecture. Its lifecycle includes the following phases:

1. Creation:

A `Session` is created through the `SessionFactory`. It is a lightweight object that represents a single-threaded unit of work.

```
Session session = sessionFactory.openSession();
```

2. Usage:

During its lifecycle, the `Session` is used to interact with the database. Operations such as saving, updating, or deleting entities are performed within a session.

```
session.beginTransaction();

// Perform database operations

session.getTransaction().commit();
```

3. Closing:

After performing the necessary operations, the `Session` should be closed to release database connections and resources.

```
session.close();
```

### Example: Managing Entity and Session Lifecycle

Here is a simple example of how you might manage both entity and session lifecycles:

```
public class EmployeeManager {

    private SessionFactory sessionFactory;

    public EmployeeManager(SessionFactory sessionFactory) {

        this.sessionFactory = sessionFactory;    }

    public void addEmployee(String name) {

        Session session = null;

        Transaction transaction = null;

        try {
            session = sessionFactory.openSession();
            transaction = session.beginTransaction();
            Employee employee = new Employee();
            employee.setName(name);
            // Persist the entity (persistent state)
            session.save(employee);
            transaction.commit(); // Entity is now synchronized with the database
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback(); // Rollback if there's an error
            }
            e.printStackTrace();
        } finally {
            if (session != null) {
                session.close(); // Closing the session
            }
        }
    }
}
```

### Hibernate Query Language (HQL)

HQL is an object-oriented query language, similar to SQL but operates on persistent objects and their properties rather than tables and columns.

https://docs.jboss.org/hibernate/orm/6.5/querylanguage/html_single/Hibernate_Query_Language.html

### Hibernate Criteria Query Language (HCQL)

HCQL is a simplified API for retrieving entities by composing criteria objects. It provides a more object-oriented way of querying the database.

Criteria queries offer a type-safe alternative to HQL, JPQL and native SQL queries.

Criteria queries are a programmatic, type-safe way to express a query. They are type-safe in terms of using interfaces and classes to represent various structural parts of a query such as the query itself, the select clause, or an order-by, etc. They can also be type-safe in terms of referencing attributes as we will see in a bit. Users of the older Hibernate org.hibernate.Criteria query API will recognize the general approach, though we believe the Jakarta Persistence API to be superior as it represents a clean look at the lessons learned from that API.

### CRUD Operations

https://github.com/anirudhagaikwad/Servlet_SpringBoot/tree/master/Practicals/HibernateJPA_JSP_APP

CRUD operations refer to the basic operations of Create, Read, Update, and Delete. In Hibernate, these can be performed using:

- Create: Use the `save()` or `persist()` method to insert records.

- Read: Use the `get()`, `load()`, or `createQuery()` methods to retrieve records.

- Update: Use the `update()` or `merge()` methods to update records.

- Delete: Use the `delete()` method to remove records.

Reading Material:

https://hibernate.org/orm/books/

1) **Java Persistence with Hibernate**

   Christian Bauer, Gavin King, and Gary Gregory

   - **ISBN 9781617290459**
   - **608 pages**
