JPA	Hibernate
JPA is described in **javax.persistence** package.	Hibernate is described in **org.hibernate** package.
It describes the handling of relational data in Java applications.	Hibernate is an Object-Relational Mapping (ORM) tool that is used to save the Java objects in the relational database system.
It is not an implementation. It is only a Java specification.	Hibernate is an implementation of JPA. Hence, the common standard which **is** given by JPA is followed by Hibernate.
It is a standard API that permits to perform database operations.	It is used in mapping Java data types with SQL data types and database tables.
As an object-oriented query language, it uses **Java Persistence Query Language (JPQL)** to execute database operations.	As an object-oriented query language, it uses **Hibernate Query Language (HQL)** to execute database operations.
To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager.	To create Session instances, it uses **SessionFactory** interface.
To make, read, and remove actions for instances of mapped entity classes, it uses **EntityManager** interface. This interface interconnect**s** with the persistence condition.	To make, read, and remove actions for instances of mapped entity classes, it uses **Session** interface. It acts as a runtime interface between a Java application and Hibernate.

JPA using Hibernate: Introduction

Definition

Java Persistence API (JPA) is a specification for accessing, persisting, and managing data between Java objects/classes and a relational database. Hibernate is an ORM (Object-Relational Mapping) framework that implements JPA, providing a way to map Java objects to database tables.

Necessity

- **Abstraction over JDBC**: JPA abstracts the boilerplate code required for database operations, making development faster and cleaner.
- **Object-Relational Mapping**: It bridges the gap between object-oriented programming and relational databases, allowing for the use of objects to represent data in a database.
- **Consistency**: JPA standardizes the way data persistence is handled, ensuring consistency across different projects and teams.

Features/Advantages

- **Simplified Database Access**: Using JPA and Hibernate, developers can perform complex database operations with simple Java methods.
- **Automatic Table Creation**: Hibernate can generate database tables based on entity mappings.
- **Lazy Loading**: Hibernate supports lazy loading, which fetches related data only when it's accessed, improving performance.
- **Caching**: Hibernate provides first-level and second-level caching to reduce database access.

Start Over Here

Student.java

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity // This marks the class as a JPA entity.
public class Student {
    @Id // This marks the field as a primary key.
    @GeneratedValue(strategy = GenerationType.IDENTITY) // This specifies that
the ID should be generated automatically.
    private Long id;
```

```
    private String name;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Main.java

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("example-unit"); // Create
EntityManagerFactory
        EntityManager em = emf.createEntityManager(); // Create EntityManager

        em.getTransaction().begin(); // Begin transaction
        Student student = new Student();
        student.setName("John Doe");
        em.persist(student); // Persist entity
        em.getTransaction().commit(); // Commit transaction

        Student foundStudent = em.find(Student.class, student.getId()); //
Find entity
        System.out.println("Found student: " + foundStudent.getName());

        em.close(); // Close EntityManager
        emf.close(); // Close EntityManagerFactory
    }
}
```

Explanation of Code

1. **Student.java**: This class is a simple JPA entity with id and name fields. The annotations @Entity, @Id, and @GeneratedValue are used to define the entity and its primary key.
2. **build.gradle**: This file contains the necessary Gradle configuration. It includes dependencies for Spring Boot, Spring Data JPA, and H2 database.
3. **application.properties**: This file contains the configuration for the H2 database, which is an in-memory database used for testing purposes.
4. **Main.java**:
    - **EntityManagerFactory emf = Persistence.createEntityManagerFactory("example-unit");**: This line creates an EntityManagerFactory using the specified persistence unit.
    - **EntityManager em = emf.createEntityManager();**: This line creates an EntityManager from the factory.
    - **em.getTransaction().begin();**: This line begins a transaction.
    - **em.persist(student);**: This line persists the Student entity.
    - **em.getTransaction().commit();**: This line commits the transaction.
    - **Student foundStudent = em.find(Student.class, student.getId());**: This line retrieves the Student entity by its ID.
    - **em.close(); emf.close();**: These lines close the EntityManager and EntityManagerFactory, respectively.

Understanding Persistence Unit in JPA

A **persistence unit** is a logical grouping that defines the set of all entity classes that are managed by EntityManager instances in an application. It is defined in the persistence.xml file, which is part of the configuration required by JPA to specify database connection details, JPA provider settings, and other properties.

Student.java

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
```

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

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

2. Managing Entities (Persisting, Updating, Deleting)

Main.java

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        // Create EntityManagerFactory
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("example-unit");
        EntityManager em = emf.createEntityManager();

        // Persisting a new student
        em.getTransaction().begin();
        Student student = new Student();
        student.setName("John Doe");
        em.persist(student);
        em.getTransaction().commit();

        // Updating an existing student
        em.getTransaction().begin();
        Student foundStudent = em.find(Student.class, student.getId());
```

```
        foundStudent.setName("Jane Doe");
        em.getTransaction().commit();

        // Deleting a student
        em.getTransaction().begin();
        em.remove(foundStudent);
        em.getTransaction().commit();

        // Close EntityManager and EntityManagerFactory
        em.close();
        emf.close();
    }
}
```

3. Querying Entities

Main.java (Continued)

```
import javax.persistence.Query;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Create EntityManagerFactory
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("example-unit");
        EntityManager em = emf.createEntityManager();

        // Persisting some students for query examples
        em.getTransaction().begin();
        Student student1 = new Student();
        student1.setName("John Doe");
        em.persist(student1);

        Student student2 = new Student();
        student2.setName("Jane Doe");
        em.persist(student2);

        Student student3 = new Student();
        student3.setName("Alice Smith");
        em.persist(student3);

        em.getTransaction().commit();

        // Querying all students
        Query query = em.createQuery("SELECT s FROM Student s");
        List<Student> students = query.getResultList();
        for (Student student : students) {
            System.out.println("Student Name: " + student.getName());
```

```
        }

        // Querying students with a specific name
        query = em.createQuery("SELECT s FROM Student s WHERE s.name =
:name");
        query.setParameter("name", "Jane Doe");
        Student singleStudent = (Student) query.getSingleResult();
        System.out.println("Found student: " + singleStudent.getName());

        // Close EntityManager and EntityManagerFactory
        em.close();
        emf.close();
    }
}
```

Explanation

1. **Entity Definition (Student.java)**:
    - The @Entity annotation marks the class as a JPA entity.
    - The @Id annotation marks the primary key field.
    - The @GeneratedValue annotation specifies the generation strategy for the primary key.
2. **Managing Entities (Main.java)**:
    - **Persisting**: Creating and saving a new entity using em.persist().
    - **Updating**: Retrieving an entity using em.find(), modifying it, and committing the transaction.
    - **Deleting**: Removing an entity using em.remove().
3. **Querying Entities (Main.java)**:
    - Using JPQL (Java Persistence Query Language) to query all students and specific students by name.
    - em.createQuery() is used to create the query.
    - query.getResultList() retrieves multiple results.
    - query.getSingleResult() retrieves a single result.

Entity Relationships in Hibernate-Specific JPA

In JPA, entity relationships represent how entities relate to each other. There are four main types of relationships:

1. **One-to-One**

2. **One-to-Many**
3. **Many-to-One**
4. **Many-to-Many**

We will use a simple Student example to explain these relationships.

1. One-to-One Relationship

A one-to-one relationship is where one entity is associated with exactly one instance of another entity.

Example: Each student has one address.

Student.java

```
import javax.persistence.*;

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

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private Address address;

    // Getters and setters
    // ...
}
```

Address.java

```
import javax.persistence.*;

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String street;
    private String city;

    // Getters and setters
    // ...
}
```

2. One-to-Many Relationship

A one-to-many relationship is where one entity is associated with multiple instances of another entity.

Example: One student can have multiple courses.

Student.java

```
import javax.persistence.*;
import java.util.List;

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

    @OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
    private List<Course> courses;

    // Getters and setters
    // ...
}
```

Course.java

```
import javax.persistence.*;

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

    @ManyToOne
    @JoinColumn(name = "student_id", nullable = false)
    private Student student;

    // Getters and setters
    // ...
}
```

3. Many-to-One Relationship

A many-to-one relationship is where multiple instances of an entity are associated with one instance of another entity.

Example: Multiple courses belong to one student (as shown in the one-to-many example above).

4. Many-to-Many Relationship

A many-to-many relationship is where multiple instances of one entity are associated with multiple instances of another entity.

Example: Students can enroll in multiple courses, and courses can have multiple students.

Student.java

```
import javax.persistence.*;
import java.util.List;

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

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;

    // Getters and setters
    // ...
}
```

Course.java

```
import javax.persistence.*;
import java.util.List;

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

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
```

```
    // Getters and setters
    // ...
}
```

Explanation

1. **One-to-One**:
    - @OneToOne annotation defines the one-to-one relationship.
    - @JoinColumn specifies the foreign key column in the owning entity.
2. **One-to-Many**:
    - @OneToMany annotation defines the one-to-many relationship.
    - mappedBy indicates the field that owns the relationship in the other entity.
3. **Many-to-One**:
    - @ManyToOne annotation defines the many-to-one relationship.
    - @JoinColumn specifies the foreign key column in the owning entity.
4. **Many-to-Many**:
    - @ManyToMany annotation defines the many-to-many relationship.
    - @JoinTable specifies the join table that holds the foreign keys of both entities.

Relationships

Necessity and Use of Different Types of Relationships in JPA

Entity relationships in JPA (and ORM in general) model real-world associations between objects in a relational database context. Understanding why and how to use these relationships is crucial for effective database design and application development.

1. One-to-One Relationship

Necessity: A one-to-one relationship is used when an entity is associated with exactly one instance of another entity. This can help in breaking down complex entities into simpler, more manageable parts.

Example Use Case: A Student entity that has an associated Address.

- **Why**: It keeps the student information separate from the address information, promoting modularity and reuse.
- **Database Schema**: There will be a foreign key in one table referring to the primary key of another table.

Example:

- **Student Table**: id, name, address_id
- **Address Table**: id, street, city

2. One-to-Many Relationship

Necessity: A one-to-many relationship is used when one entity is associated with multiple instances of another entity. This is common in hierarchical structures.

Example Use Case: A Student entity that has multiple Course entities.

- **Why**: It allows one student to be enrolled in multiple courses.
- **Database Schema**: The Course table will have a foreign key referring to the Student table.

Example:

- **Student Table**: id, name
- **Course Table**: id, course_name, student_id

3. Many-to-One Relationship

Necessity: A many-to-one relationship is essentially the reverse of a one-to-many relationship. It's used when multiple instances of an entity are associated with one instance of another entity.

Example Use Case: Multiple Course entities belong to one Student.

- **Why**: It allows us to track which student is taking which courses.
- **Database Schema**: Similar to the one-to-many relationship, the Course table has a foreign key to the Student table.

Example:

- **Student Table**: id, name
- **Course Table**: id, course_name, student_id

4. Many-to-Many Relationship

Necessity: A many-to-many relationship is used when multiple instances of one entity are associated with multiple instances of another entity. This is common in many real-world scenarios.

Example Use Case: Students enrolling in multiple courses, and each course having multiple students.

- **Why**: It models complex relationships where entities are interlinked.
- **Database Schema**: Requires a join table (also known as a junction table) to manage the associations.

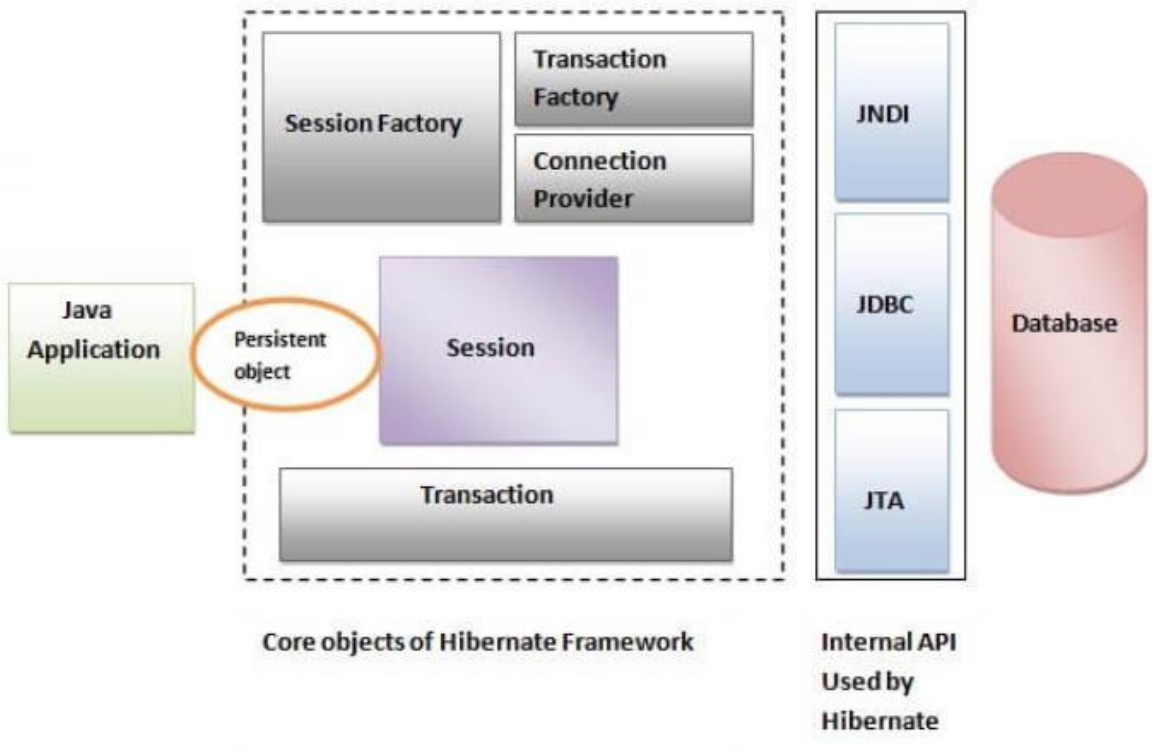
Example:

- **Student Table**: id, name
- **Course Table**: id, course_name
- **Student_Course Table**: student_id, course_id

How Relationships Work with Databases

1. **One-to-One**:
    - Uses a foreign key in the Student table that references the primary key in the Address table.
    - Ensures that each student can only have one address and vice versa.
2. **One-to-Many**:
    - Uses a foreign key in the Course table that references the primary key in the Student table.
    - Allows multiple courses to be linked to a single student.
3. **Many-to-One**:
    - Similar to one-to-many but from the perspective of the many-side entity.
    - Multiple courses refer to a single student, enforcing that each course is linked to a specific student.
4. **Many-to-Many**:
    - Uses a join table that holds foreign keys from both Student and Course tables.
    - Facilitates the linking of multiple students to multiple courses and vice versa.

Hibernate Architecture



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

Elements of Hibernate Architecture

For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follow

SessionFactory

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

Session

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

Transaction

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

ConnectionProvider

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

TransactionFactory

It is a factory of Transaction. It is optional.

Spring Core Introduction / Overview

Spring Core is the foundation of the Spring Framework, providing fundamental features for dependency injection and application context management. It is essential for creating scalable, maintainable, and modular Java applications. Here, we'll cover the key concepts, components, and features of Spring Core.

Definition

Spring Core: The core module of the Spring Framework that provides the fundamental features like Dependency Injection (DI) and Inversion of Control (IoC). It is responsible for managing the lifecycle and configuration of application objects.

Key Concepts

1. **Dependency Injection (DI)**: A design pattern used to implement IoC, where the control of object creation and dependency management is given to a container or framework. This promotes loose coupling and easier testing.
2. **Inversion of Control (IoC)**: A principle where the control flow of the application is inverted, and the control is given to a container or framework. This container manages the lifecycle and configuration of objects.

3. **Application Context**: The central interface in Spring for providing configuration information to the application. It is an extension of the BeanFactory interface, providing more enterprise-specific functionality.

Components and Features

1. **Beans**: Objects that are managed by the Spring IoC container. They are created, configured, and managed by the Spring container.
2. **BeanFactory**: The basic container that provides the configuration framework and basic functionality for managing beans.
3. **ApplicationContext**: A more advanced container that extends BeanFactory and adds more enterprise-specific features such as event propagation, declarative mechanisms to create a bean, and various means to look up.
4. **Configuration Metadata**: Metadata used by the Spring container to configure beans. This metadata can be provided in various forms, such as XML, annotations, or Java code.

Why Spring Core?

- **Loose Coupling**: By using DI and IoC, Spring promotes loose coupling between the components, making the application more modular and easier to maintain.
- **Testability**: Spring makes it easier to write unit tests by allowing you to inject mock dependencies.
- **Modularity**: Encourages modular development by promoting separation of concerns.
- **Integration**: Spring integrates well with other frameworks and technologies, making it suitable for enterprise application development.

Example with Explanation

Let's create a simple example to understand how Spring Core works. We will create a Student service that depends on a StudentRepository.

1. Maven Configuration (pom.xml)

Add the following dependencies in your pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.9</version>
  </dependency>
</dependencies>
```

2. Spring Configuration (Java-based)

We will use Java-based configuration to define our beans.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public StudentRepository studentRepository() {
        return new StudentRepository();
    }

    @Bean
    public StudentService studentService() {
        return new StudentService(studentRepository());
    }
}
```

3. Service and Repository Classes

```java
// Student.java
public class Student {
    private int id;
    private String name;

    // Constructors, getters, and setters
}

// StudentRepository.java
public class StudentRepository {
    public Student getStudentById(int id) {
        return new Student(id, "John Doe");
    }
}

// StudentService.java
public class StudentService {
    private final StudentRepository studentRepository;

    public StudentService(StudentRepository
studentRepository) {
```

```
        this.studentRepository = studentRepository;
    }

    public Student findStudent(int id) {
        return studentRepository.getStudentById(id);
    }
}
```

4. Main Class to Run the Application

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        StudentService studentService = context.getBean(StudentService.class);

        Student student = studentService.findStudent(1);
        System.out.println("Student Name: " + student.getName());
    }
}
```

Explanation

1. **Configuration**: AppConfig class is marked with @Configuration indicating that it is a source of bean definitions. The @Bean annotation tells Spring that the method will return an object that should be registered as a bean in the Spring application context.
2. **Dependency Injection**: The StudentService class is dependent on StudentRepository. In AppConfig, we define the beans and inject the dependencies. This allows the StudentService to use the StudentRepository without needing to instantiate it manually.
3. **Application Context**: In the Main class, we use AnnotationConfigApplicationContext to load the Spring context from the AppConfig class. The context is used to get the StudentService bean and call its method.