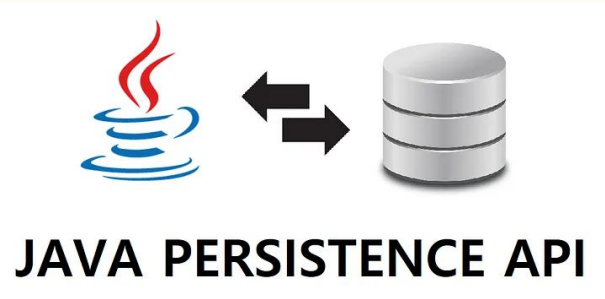
Module 6:

Introduction to JPA and Hibernate

- Prof.Anamika Dhawan

Java Persistence API (JPA)



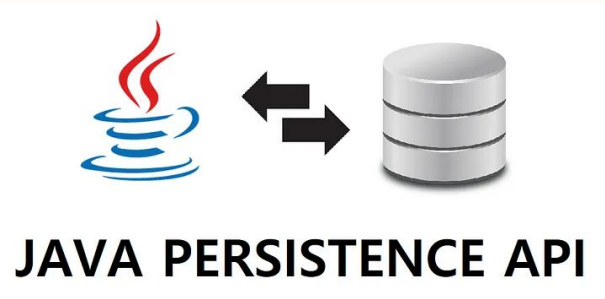
The Java Persistence API (JPA) is a standard for managing relational data in Java applications.

It is part of the Java EE (Enterprise Edition) and Jakarta EE platforms, offering a powerful and flexible framework for mapping Java objects to database tables and vice versa.

JPA provides a unified approach to object-relational mapping (ORM), making it easier to handle database interactions within Java applications.

This article will explore JPA in-depth, covering its architecture, features, usage, best practices, and comparisons with related technologies like Hibernate.

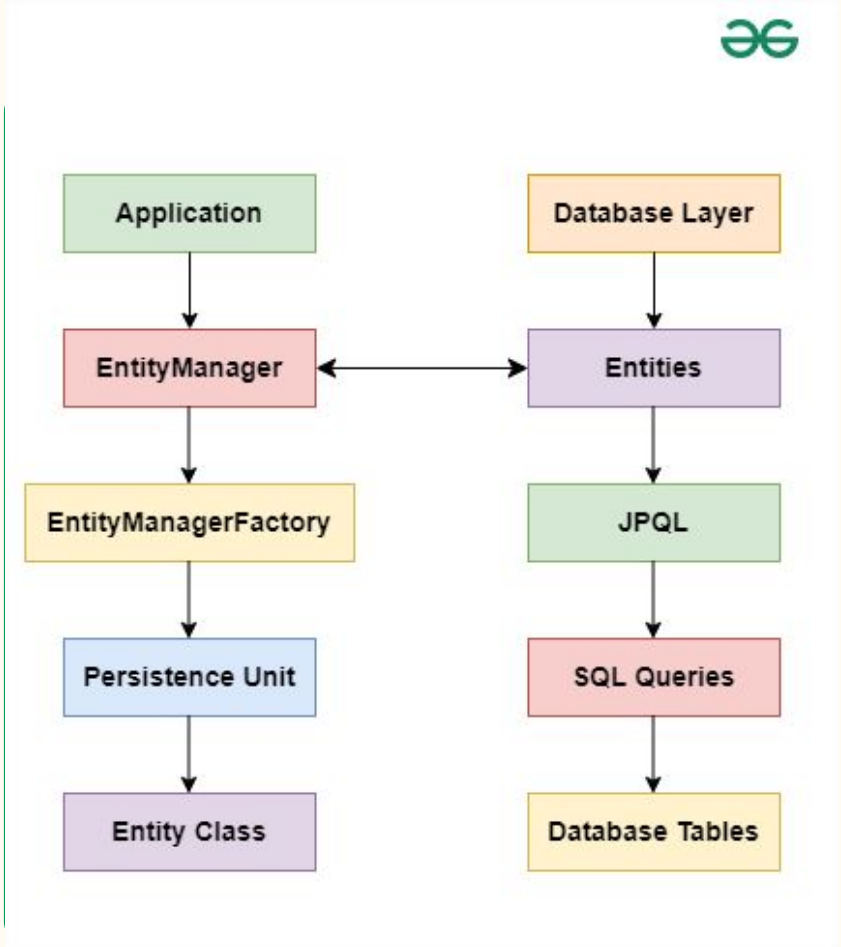
Introduction to JPA



JPA abstracts the complexities of database interactions, allowing you to work with Java objects without dealing directly with SQL queries. This abstraction layer promotes a more object-oriented approach to database management, improving code readability, maintainability, and scalability.

JPA Class-level Architecture:

JPA has 6 layers in its class-level architecture:



Core Components of JPA

1. Entity:

An entity in JPA is a lightweight, persistent domain object that represents a table in a relational database. Each instance of an entity corresponds to a row in the table. Entities are typically annotated with @Entity, and fields are mapped to table columns using annotations like @Column, @Id, and @GeneratedValue.

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

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

    private String name;
    private String email;

    // Getters and setters
}
```

Annotation	Description
@Entity	Used to tell Java to treat this class as the Entity (or model) class
@Id	Sets the following variable as the primary key of the table in the database to which this Entity class is mapped
@GeneratedValue(strategy = GenerationType.IDENTITY)	Generates unique, auto-incrementing values for the variable mapped to the primary key of the table in the database
@Column	Specifies to the Java class that the following variable is mapped to the corresponding position column in the table of the mapped database

Core Components of JPA

2. EntityManager:

The EntityManager interface is the primary interface used for interacting with the persistence context. It provides methods for creating, reading, updating, and deleting entities, as well as for executing queries.

```
import javax.persistence.Persistence;

public class UserService {
    private EntityManagerFactory emf = Persistence.createEnti

    public void createUser(String name, String email) {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        User user = new User();
        user.setName(name);
        user.setEmail(email);

        em.persist(user);
        em.getTransaction().commit();
        em.close();
```

Core Components of JPA

3. Persistence Unit:

A persistence unit is a logical grouping of related entity classes and configuration metadata. It is defined in the `persistence.xml` file, which specifies connection properties, entity classes, and other settings.

Example (`persistence.xml`):

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
version="2.2">
   <persistence-unit name="my-persistence-unit">
       <class>com.example.User</class>
       <properties>
           <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/mydb"/>
           <property name="javax.persistence.jdbc.user" value="root"/>
           <property name="javax.persistence.jdbc.password"
value="password"/>
           <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
           <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect"/>
       </properties>
   </persistence-unit>
</persistence>
```

Core Components of JPA

4. EntityManagerFactory:

The EntityManagerFactory is a factory for EntityManager instances. The createEntityManager() method creates a new EntityManager instance

Core Components of JPA

5. Query Language:

JPA introduces JPQL (Java Persistence Query Language), an object-oriented query language similar to SQL but operating on entity objects rather than database tables.

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import java.util.List;

public class UserService {
   private EntityManagerFactory emf =
Persistence.createEntityManagerFactory("my-persistence-unit");

   public List<User> getAllUsers() {
       EntityManager em = emf.createEntityManager();
       TypedQuery<User> query = em.createQuery("SELECT u FROM User u", User.class);
       List<User> users = query.getResultList();
       em.close();
       return users;
   }
}
```

Core Components of JPA

6. Database Layer
The Database Layer contains the relational database that stores all the data of the application. The tables in the database are mapped to entity classes in the Java application.

SQL Code Example for creating a database and a table in it:

```
CREATE DATABASE mydb;

USE mydb;

CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE
);
```

Key features of JPA

1. Object-Relational Mapping (ORM):

JPA simplifies the mapping of Java objects to relational database tables, eliminating the need for boilerplate SQL code.

2. Annotations and XML Configuration:

JPA supports both annotations and XML-based configuration, allowing you to choose the approach that best suits your project requirements.

3. Transaction Managemet:

JPA provides built-in support for transaction management, ensuring data consistency and integrity.

4. Caching:

JPA supports caching mechanisms to improve performance by reducing the number of database accesses.

Key features of JPA

5. Lifecycle Callbacks:

JPA provides lifecycle callback methods (e.g., `@PrePersist`, `@PostPersis`, `@PreUpdate`, `@PostUpdate`) that allow you to perform actions before or after certain entity operations.

6. Inheritance Mapping:

JPA supports inheritance mapping strategies (e.g., `SINGLE TABLE`, `TABLE PER CLASS`, `JOINED`) to represent inheritance hierarchies in the database.

7. Relationships:

JPA simplifies the management of entity relationships (e.g., `OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany`), making it easier to model complex data structures.

Java Persistence API vs. Hibernate

Hibernate is a popular ORM framework that predates JPA. It implements the JPA specification and provides additional features and capabilities. Here's a comparison between JPA and Hibernate:

1. Standard vs. Implementation:

- **JPA**: JPA is a specification that defines a standard approach for ORM in Java.
- **Hibernate**: Hibernate is an implementation of the JPA specification, offering additional features beyond the standard.

Java Persistence API vs. Hibernate

2. Vendor Neutrality:

JPA: Being a specification, JPA can be implemented by various vendors, providing flexibility in choosing the underlying implementation (e.g., Hibernate, EclipseLink).

Hibernate: Hibernate is a specific implementation, which can be used as a JPA provider or standalone with its native API.

3. Features:

JPA: JPA provides a standardized feature set for ORM.

Hibernate: Hibernate extends JPA with additional features like caching, auditing, and advanced query capabilities (e.g., Criteria API, HQL).

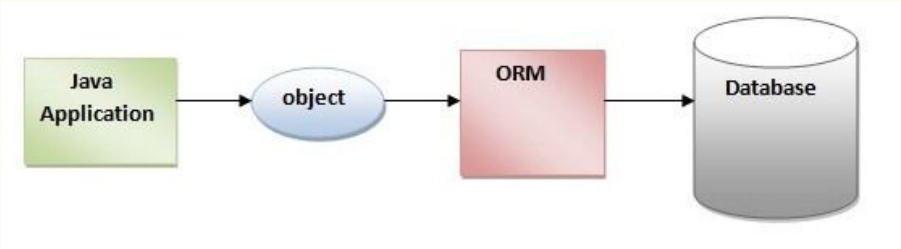
Java Persistence API vs. Hibernate

4. Community and Support:

JPA: As a standard, JPA has broad support across the Java ecosystem, with extensive documentation and community resources.

Hibernate: Hibernate has a large and active community, with extensive documentation, forums, and commercial support options.

Overview of Hibernate



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

ORM Tool

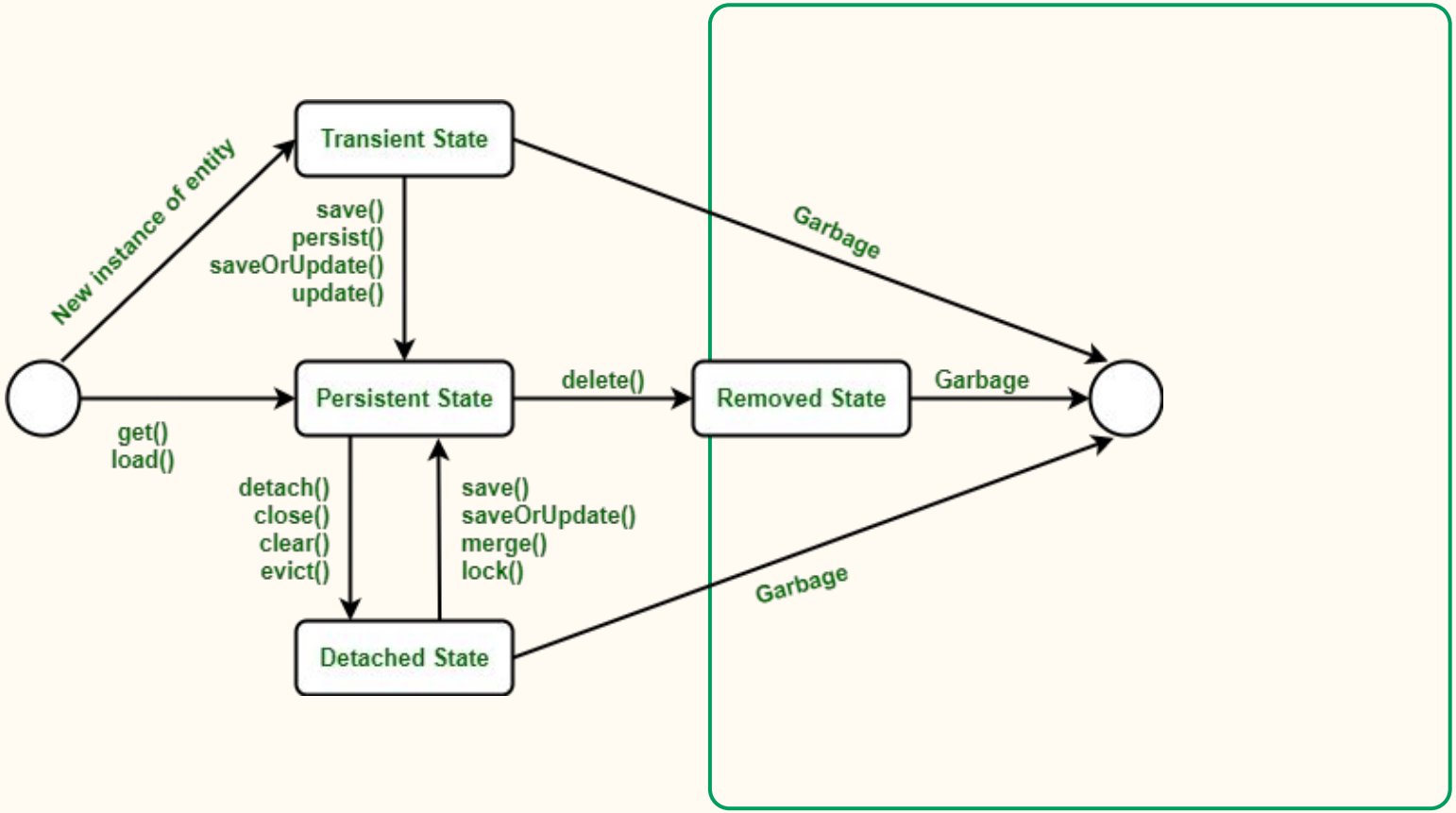
An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.

Hibernate Architecture

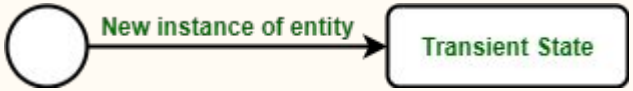
In Hibernate, we can either create a new object of an entity and store it into the database, or we can fetch the existing data of an entity from the database. These entity is connected with the lifecycle, and each object of entity passes through the various stages of the lifecycle.

There are mainly four states of the Hibernate Lifecycle :

1. Transient State
2. Persistent State
3. Detached State
4. Removed State


Transient State
New instance of entity
save()
persist()
saveOrUpdate()
update()
Garbage
Persistent State
delete()
Removed State
Garbage
get()
load()
detach()
close()
clear()
evict()
save()
saveOrUpdate()
merge()
lock()
Garbage
Detached State

1.Transient State



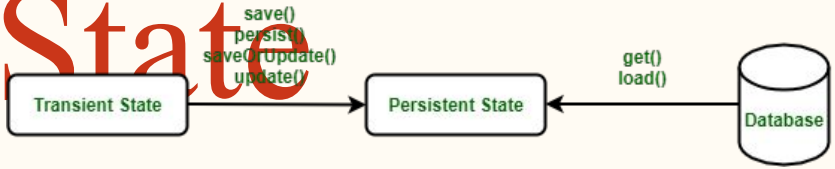

```
//Here, The object arrives in the transient state.
Employee e = new Employee();
e.setId(21);
e.setFirstName("Neha");
e.setMiddleName("Shri");
e.setLastName("Rudra");
```

The transient state is the first state of an entity object. When we instantiate an object of a POJO class using the new operator then the object is in the transient state. This object is not connected with any hibernate session. As it is not connected to any Hibernate Session, So this state is not connected to any database table. So, if we make any changes in the data of the POJO Class then the database table is not altered. Transient objects are independent of Hibernate, and they exist in the **heap memory**.

There are two layouts in which transient state will occur as follows:

1. When objects are generated by an application but are not connected to any session.
2. The objects are generated by a closed session.

State 2: Persistent State




```
// Transient State
Employee e = new Employee("Neha Shri Rudra", 21, 180103);


//Persistent State
session.save(e);
```

Once the object is connected with the Hibernate Session then the object moves into the Persistent State. So, there are two ways to convert the Transient State to the Persistent State :

Using the hibernated session, save the entity object into the database table.

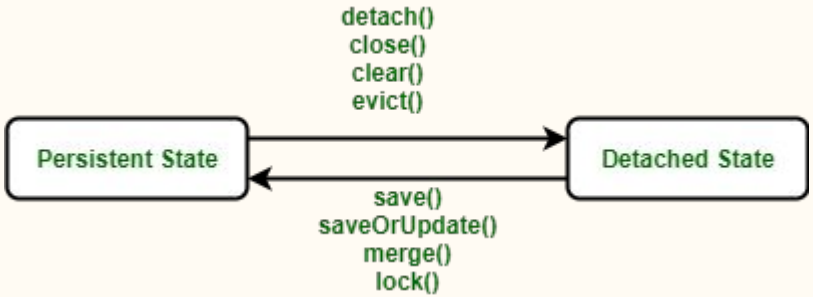
Using the hibernated session, load the entity object into the database table.
In this state. each object represents one row in the database table. Therefore, if we make any changes in the data then hibernate will detect these changes and make changes in the database table.

State 2: Persistent State

Following are the methods given for the persistent state:

- session.persist(e);
- session.save(e);
- session.saveOrUpdate(e);
- session.update(e);
- session.merge(e);
- session.lock(e);

3. Detached State




```
// Transient State
Employee e = new Employee("Neha Shri Rudra", 21, 180103);

// Persistent State
session.save(e);


// Detached State
session.close();
```

For converting an object from Persistent State to Detached State, we either have to close the session or we have to clear its cache. As the session is closed here or the cache is cleared, then any changes made to the data will not affect the database table. Whenever needed, the detached object can be reconnected to a new hibernate session. To reconnect the detached object to a new hibernate session, we will use the following methods as follows:

- merge()
- update()
- load()
- refresh()
- save()
- update()

Following are the methods used for the detached state :

- session.detach(e);
- session.evict(e);
- session.clear();
- session.close();

4. Removed State


```
// Java Pseudo code to Illustrate Remove State

// Transient State
Employee e = new Employee();
Session s = sessionfactory.openSession();
e.setId(01);


// Persistent State
session.save(e)

// Removed State
session.delete(e);
```

In the hibernate lifecycle it is the last state. In the removed state, when the entity object is deleted from the database then the entity object is known to be in the removed state. It is done by calling the ***delete() operation***. As the entity object is in the removed state, if any change will be done in the data will not affect the database table.