Unit 3 JPA

Java Persistence API (JPA) is a specification in Java for managing relational data in applications. It provides a standard way to map Java objects to database tables and to persist data. JPA allows developers to work with object-oriented models without needing to write a lot of boilerplate SQL code, thus simplifying data management tasks.

Hibernate is a popular implementation of the JPA specification. It serves as an Object-Relational Mapping (ORM) tool that bridges the gap between the object-oriented world of Java and the relational world of databases. Hibernate automates the data persistence process, making it easier to handle database operations within Java applications.

Key Features of JPA with Hibernate

1. **Object-Relational Mapping (ORM)**:
    - **Entities**: Map Java classes to database tables.
    - **Attributes**: Map class fields to table columns.
    - **Relationships**: Define relationships between entities using annotations like `@OneToMany`, `@ManyToOne`, `@OneToOne`, and `@ManyToMany`.
2. **CRUD Operations**:
    - Simplify Create, Read, Update, and Delete operations using repository interfaces like `JpaRepository`.
3. **Querying**:
    - Use JPQL (Java Persistence Query Language) to write database queries in an object-oriented way.
    - Use Criteria API to build queries programmatically.
4. **Transaction Management**:
    - Manage database transactions to ensure data consistency and integrity.
5. **Caching**:
    - Improve application performance by using first-level and second-level caches.
6. **Validation**:
    - Ensure data integrity with built-in validation support using annotations like `@NotNull`, `@Size`, etc.

Why Use JPA with Hibernate?

1. **Simplifies Data Management**: Abstracts the complexity of JDBC and SQL, allowing developers to focus on business logic.
2. **Reduces Boilerplate Code**: Automatically handles the mapping between Java objects and database tables, reducing the amount of code needed for database interactions.
3. **Scalability and Performance**: Efficiently manages data retrieval and storage, with support for caching and lazy loading to optimize performance.
4. **Vendor Independence**: By using JPA, you can switch between different JPA providers without changing your data access code, giving you flexibility and portability.
5. **Community and Support**: Hibernate, being a widely-used framework, has extensive documentation, a large community, and numerous resources for learning and troubleshooting.

Getting Started

To start using JPA with Hibernate, you need to:

1. **Set Up Dependencies**: Add the necessary dependencies to your project (e.g., via Maven or Gradle).
2. **Configure Persistence**: Set up your database connection and Hibernate properties.
3. **Define Entities**: Create Java classes annotated with JPA annotations to map to database tables.
4. **Create Repositories**: Define repository interfaces to handle CRUD operations.
5. **Perform Operations**: Use the repository methods to perform database operations.

Entity

In JPA (Java Persistence API), an **entity** 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 defined using Java classes and are annotated to map the class properties to the corresponding table columns.

Entity Basics

An entity in JPA is a simple Java class that is annotated with `@Entity`. The primary key of the entity is specified using the `@Id` annotation.

Example of a Basic Entity

```
import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String name;

    private String email;

    // Getters and Setters

}
```

In this example:

- @Entity: Marks the class as a JPA entity.
- @Id: Specifies the primary key of the entity.
- @GeneratedValue: Specifies how the primary key should be generated.

Types of Entities

Entities in JPA can be categorized based on their relationships and inheritance strategies. Here are the main types:

1. **Basic Entity**:
   - A simple entity with fields that map directly to table columns.
2. **Embeddable Entity**:
   - An entity that is not an independent table but can be embedded in other entities using the @Embeddable and @Embedded annotations.
   - **Example**:

```
java
Copy code
import javax.persistence.Embeddable;

@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;

    // Getters and Setters
}

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

    @Embedded
    private Address address;

    // Getters and Setters
}
```

3. **Entity with Relationships**:
   - Entities can have relationships with other entities using annotations such as @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany.
   - **Example**:

```
java
Copy code
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import java.util.List;
```

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

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

   // Getters and Setters
}

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

   @ManyToOne
   private Department department;

   // Getters and Setters
}
```

4. **Inheritance Entities**:
   - JPA supports inheritance, allowing entities to inherit properties from a base class. There are three inheritance strategies: SINGLE_TABLE, JOINED, and TABLE_PER_CLASS.
   - **Example**:

```
java
Copy code
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "employee_type")
public abstract class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;

   // Getters and Setters
}

@Entity
@DiscriminatorValue("FT")
public class FullTimeEmployee extends Employee {
   private double salary;
```

```
    // Getters and Setters
}

@Entity
@DiscriminatorValue("PT")
public class PartTimeEmployee extends Employee {
    private double hourlyWage;

    // Getters and Setters
}
```

Entities are the cornerstone of JPA, providing a way to map Java objects to database tables. By leveraging entities and their various types and relationships, developers can model complex data structures and relationships within their applications. Understanding the different types of entities and their uses is crucial for effectively using JPA in your Java applications.

Managing Entity

Managing entities in JPA involves several key tasks, including persisting new entities, updating existing ones, retrieving entities, and deleting them. Additionally, handling transactions and understanding the lifecycle of entities are crucial aspects of entity management.

Persisting Entities

To save a new entity to the database, use the EntityManager or a repository provided by Spring Data JPA.

Using EntityManager

java
Copy code

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.Transactional;

public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void createUser(User user) {
        entityManager.persist(user);
    }
}
```

Using Spring Data JPA Repository

java
Copy code

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
```

```
    @Autowired
    private UserRepository userRepository;

    public void createUser(User user) {
        userRepository.save(user);
    }
}
```

Updating Entities

Updating an entity involves retrieving the entity, modifying its fields, and then saving the changes.

Using EntityManager

java
Copy code

```
@Transactional
public void updateUser(Long id, String newName) {
    User user = entityManager.find(User.class, id);
    if (user != null) {
        user.setName(newName);
        entityManager.merge(user);
    }
}
```

Using Spring Data JPA Repository

java
Copy code

```
public void updateUser(Long id, String newName) {
    User user = userRepository.findById(id).orElseThrow(() -> new EntityNotFoundException("User not found"));
    user.setName(newName);
    userRepository.save(user);
}
```

Retrieving Entities

Entities can be retrieved using the EntityManager or repository methods.

Using EntityManager

java
Copy code

```
public User getUser(Long id) {
    return entityManager.find(User.class, id);
}
```

Using Spring Data JPA Repository

java
Copy code

```
public User getUser(Long id) {
    return userRepository.findById(id).orElse(null);
}
```

Deleting Entities

To delete an entity, it must be retrieved first, and then removed using the EntityManager or repository methods.

Using EntityManager

java
Copy code
```
@Transactional
public void deleteUser(Long id) {
    User user = entityManager.find(User.class, id);
    if (user != null) {
        entityManager.remove(user);
    }
}
```

Using Spring Data JPA Repository

java
Copy code
```
public void deleteUser(Long id) {
    userRepository.deleteById(id);
}
```

Transactions

Transactions ensure data consistency and integrity. In JPA, transactions can be managed manually or automatically.

Manual Transaction Management

java
Copy code
```
import javax.persistence.EntityTransaction;

public void manualTransaction() {
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
        // Perform operations
        transaction.commit();
    } catch (Exception e) {
        transaction.rollback();
    }
}
```

Automatic Transaction Management with Spring

java
Copy code
```
import org.springframework.transaction.annotation.Transactional;

@Transactional
public void performTransactionalOperation() {
    // Operations that need to be in a transaction
}
```

Entity Lifecycle

Understanding the lifecycle states of entities is essential for effective entity management:

1. **New**: The entity is created but not yet associated with the persistence context.

2. **Managed**: The entity is associated with the persistence context and synchronized with the database.
3. **Detached**: The entity is no longer associated with the persistence context.
4. **Removed**: The entity is marked for removal from the database.

Example: Entity Lifecycle

java
Copy code

```
public void entityLifecycleExample() {
    User user = new User(); // New state
    entityManager.persist(user); // Managed state
    entityManager.detach(user); // Detached state
    entityManager.remove(user); // Removed state
}
```

Best Practices

1. **Use DTOs**: Use Data Transfer Objects to transfer data between layers and avoid exposing entities directly.
2. **Validation**: Ensure data integrity with validation annotations (@NotNull, @Size, etc.).
3. **Batch Processing**: Use batch processing for bulk operations to optimize performance.
4. **Lazy Loading**: Be cautious with lazy loading to avoid LazyInitializationException.
5. **Caching**: Leverage second-level caching for read-heavy applications.

By following these practices and understanding the core concepts, you can efficiently manage entities in your JPA application.

Querying Entites

Querying entities in JPA involves using JPQL (Java Persistence Query Language), Criteria API, and native SQL queries. Each of these methods provides different ways to retrieve data from the database based on the specific requirements.

1. JPQL (Java Persistence Query Language)

JPQL is an object-oriented query language similar to SQL but operates on entities instead of database tables.

Basic JPQL Queries

java
Copy code

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import java.util.List;

public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> getAllUsers() {
```

```
        String jpql = "SELECT u FROM User u";
        TypedQuery<User> query = entityManager.createQuery(jpql, User.class);
        return query.getResultList();
    }

    public User getUserByEmail(String email) {
        String jpql = "SELECT u FROM User u WHERE u.email = :email";
        TypedQuery<User> query = entityManager.createQuery(jpql, User.class);
        query.setParameter("email", email);
        return query.getSingleResult();
    }
}
```

JPQL with Named Queries

Named queries are defined in the entity class using the @NamedQuery annotation.

```
java
Copy code
import javax.persistence.Entity;
import javax.persistence.NamedQuery;
import javax.persistence.Id;

@Entity
@NamedQuery(name = "User.findByName", query = "SELECT u FROM User u WHERE u.name = :name")
public class User {
    @Id
    private Long id;
    private String name;
    private String email;

    // Getters and Setters
}

public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> getUsersByName(String name) {
        TypedQuery<User> query = entityManager.createNamedQuery("User.findByName", User.class);
        query.setParameter("name", name);
        return query.getResultList();
    }
}
```

2. Criteria API

The Criteria API allows for the creation of queries in a type-safe and dynamic way using Java objects.

Basic Criteria Query

```
java
Copy code
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import java.util.List;
```

```
public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> getAllUsers() {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> cq = cb.createQuery(User.class);
        Root<User> rootEntry = cq.from(User.class);
        CriteriaQuery<User> all = cq.select(rootEntry);

        TypedQuery<User> query = entityManager.createQuery(all);
        return query.getResultList();
    }

    public List<User> getUsersByName(String name) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> cq = cb.createQuery(User.class);
        Root<User> user = cq.from(User.class);
        cq.where(cb.equal(user.get("name"), name));

        TypedQuery<User> query = entityManager.createQuery(cq);
        return query.getResultList();
    }
}
```

3. Native SQL Queries

Native queries allow you to write raw SQL queries and map the results to entity classes.

Basic Native Query

java
Copy code

```
import javax.persistence.Query;

public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> getAllUsers() {
        String sql = "SELECT * FROM User";
        Query query = entityManager.createNativeQuery(sql, User.class);
        return query.getResultList();
    }

    public User getUserByEmail(String email) {
        String sql = "SELECT * FROM User WHERE email = :email";
        Query query = entityManager.createNativeQuery(sql, User.class);
        query.setParameter("email", email);
        return (User) query.getSingleResult();
    }
}
```

4. Using Spring Data JPA Repositories

Spring Data JPA simplifies querying with predefined repository methods and derived query methods.

Derived Query Methods

java
Copy code
```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);
    User findByEmail(String email);
}
```

Custom Queries with @Query

java
Copy code
```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.name = :name")
    List<User> findUsersByName(@Param("name") String name);
}
```

Advanced Querying Techniques

Pagination and Sorting

java
Copy code
```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

public interface UserRepository extends JpaRepository<User, Long> {
    Page<User> findAll(Pageable pageable);
}
```

Specifications (Dynamic Queries)

java
Copy code
```
import org.springframework.data.jpa.domain.Specification;

public class UserSpecifications {
    public static Specification<User> hasName(String name) {
        return (user, cq, cb) -> cb.equal(user.get("name"), name);
    }

    public static Specification<User> hasEmail(String email) {
        return (user, cq, cb) -> cb.equal(user.get("email"), email);
    }
}

import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

public interface UserRepository extends JpaRepository<User, Long>, JpaSpecificationExecutor<User> {
}
```

Querying entities in JPA can be done using JPQL, Criteria API, native SQL queries, and Spring Data JPA repositories. Each method has its advantages and use cases. JPQL provides an object-oriented approach, Criteria API offers type safety and dynamic query construction, native SQL gives direct control over SQL execution, and Spring Data JPA simplifies query generation with repository abstractions. Understanding and using these querying techniques

effectively will enable you to retrieve and manipulate data efficiently in your JPA applications.

Entity Relationship

Entity relationships in JPA are crucial for modeling real-world data and interactions between different entities. JPA provides several annotations to define these relationships, including @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany. Each of these annotations can be used to specify the nature of the relationship between entities.

Types of Entity Relationships

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

1. One-to-One Relationship

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

Example

java
Copy code

```
import javax.persistence.*;

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

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

    // Getters and Setters
}

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

    @OneToOne(mappedBy = "address")
    private User user;

    // Getters and Setters
```

```
}
```

2. One-to-Many Relationship

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

Example

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

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

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Employee> employees;

    // Getters and Setters
}

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

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

    // Getters and Setters
}
```

3. Many-to-One Relationship

A many-to-one relationship is the inverse of a one-to-many relationship where multiple instances of one entity are associated with one instance of another entity. This is typically represented using the `@ManyToOne` annotation on the child entity.

Example

The `Employee` class above with the `@ManyToOne` annotation demonstrates a many-to-one relationship with the `Department` entity.

4. Many-to-Many Relationship

A many-to-many relationship is a relationship where multiple instances of one entity are associated with multiple instances of another entity. This usually involves a join table to hold the relationships.

Example

java
Copy code

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

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

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

    // Getters and Setters
}

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

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

    // Getters and Setters
}
```

Fetch Types

JPA provides two types of fetching strategies for relationships: EAGER and LAZY.

- **EAGER**: The associated entity is loaded immediately.
- **LAZY**: The associated entity is loaded on demand.

Example

java
Copy code

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

Cascade Types

Cascade types define the operations that should be cascaded to the associated entities.

- **ALL**: All operations
- **PERSIST**: Persist operation
- **MERGE**: Merge operation
- **REMOVE**: Remove operation
- **REFRESH**: Refresh operation
- **DETACH**: Detach operation

Example

java
Copy code

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

Orphan Removal

Orphan removal allows you to remove child entities when they are no longer referenced by the parent entity.

Example

java
Copy code

```
@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Employee> employees;
```

Entity relationships in JPA are essential for representing complex data models. By using annotations like @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany, you can define how entities interact with each other. Understanding fetch types, cascade types, and orphan removal is crucial for effectively managing these relationships and ensuring data consistency in your application.