LESSON 23
MySQL & JPA Advanced

WEEK 05

Inheritance in JPA Entities

Introduction to Inheritance in JPA Entities

- **Introduction to Inheritance in JPA Entities**
- **What is Inheritance in JPA?**:
  - JPA supports inheritance to model hierarchical entity classes, mapping Java OO inheritance to relational databases.
  - Allows subclasses to inherit fields and relationships from a superclass.
- **Inheritance Strategies**:
  - @Inheritance(strategy = InheritanceType.SINGLE_TABLE): All classes in one table with discriminator.
  - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS): Separate table per concrete class.
  - @Inheritance(strategy = InheritanceType.JOINED): Separate table for superclass and each subclass (1:1 relationship via shared primary key).
- **Focus on 1:1 Inheritance**:
  - Refers to JOINED strategy, where subclass tables link 1:1 to superclass table using shared PK.
- **Reference**: JPA Inheritance Specification

JOINED Inheritance Strategy (1:1 Mapping)

- **Definition**:
  - Superclass has its own table; each subclass has a separate table with only subclass-specific fields.
  - Subclass tables reference superclass table via shared primary key (1:1 relationship).
- **Annotations**:
  - @Inheritance(strategy = InheritanceType.JOINED) on superclass.
  - @PrimaryKeyJoinColumn optional for customizing join column.
- **Database Structure**:
  - Superclass table: Common fields + PK.
  - Subclass table: Subclass fields + PK (foreign key to superclass PK).
- **Explanation**:
  - Queries join tables as needed; supports polymorphism (e.g., querying superclass returns mixed subclass instances).

Example of 1:1 Inheritance in JPA

- **Generated Tables**:
  - person: id (PK), name.
  - student: id (PK/FK to person.id), major.
  - teacher: id (PK/FK to person.id), subject.
- **Explanation**:
  - Inserting a Student creates rows in both person and student tables with same id.
  - Query: SELECT p FROM Person p joins tables to fetch mixed Student/Teacher instances.

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and setters
}

@Entity
public class Student extends Person {
    private String major;
    // Getters and setters
}

@Entity
public class Teacher extends Person {
    private String subject;
    // Getters and setters
}
```

Advantages and Disadvantages of 1:1 Inheritance

- **Advantages**:
  - Normalized database: No redundant fields; easy to add new subclasses.
  - Supports polymorphism: Queries on superclass return subclass instances.
  - Efficient for reads on specific subclasses (no unnecessary joins).
- **Disadvantages**:
  - Performance overhead: Joins required for superclass queries.
  - Complex inserts/updates: Multiple tables involved.
  - Not suitable for deep hierarchies due to join complexity.
- **When to Use**:
  - When normalization is important and hierarchies are not too deep.

Best Practices for 1:1 Inheritance in JPA

- **Key Practices**:
  - Use @DiscriminatorColumn if needed for explicit type discrimination (though optional in JOINED).
  - Optimize queries with Fetch Joins to avoid N+1 issues.
  - Index join columns for better performance.
  - Test polymorphism: Ensure repositories handle superclass queries correctly.
- **Common Pitfalls**:
  - Overusing joins in deep hierarchies leading to slow queries.
  - Forgetting to generate IDs in superclass.

```
@Query("SELECT p FROM Person p JOIN FETCH p WHERE p.id = :id")
Person findByIdWithFetch(@Param("id") Long id);
```

JPA Entity Graphs for Dynamic Fetching

JPA Entity Graphs for Dynamic Fetching

- **What is an Entity Graph?**:
  - Allows dynamic specification of which relationships to fetch (eager or lazy) for a query.
  - Overrides default @FetchType settings in entity mappings.
- **Types**:
  - @NamedEntityGraph: Defined statically in entity class.
  - Dynamic Entity Graph: Created programmatically via EntityManager.
- **Explanation**:
  - Fetches department eagerly for findAll(), overriding LAZY.
  - Avoids N+1 issues by specifying related data in a single query.

JPA Entity Graphs for Dynamic Fetching

```java
@Entity
@NamedEntityGraph(name = "Student.WithDepartmentAndCourses", attributeNodes = {
        @NamedAttributeNode("courses"),
        @NamedAttributeNode("department")
})
@NamedEntityGraph(name = "Student.WithCourses", attributeNodes = {
        @NamedAttributeNode("courses")
})
@NamedEntityGraph(name = "Student.WithDepartment", attributeNodes = {
        @NamedAttributeNode("department")
})
@Table(name = "students")
@FilterDef(name = "AvailableStudents", parameters = @ParamDef(name = "deleted", type = Boolean.class))
@Filter(name = "AvailableStudents", condition = "deleted = :deleted")
public class Student {
```

JPA Entity Graphs for Dynamic Fetching

```java
// Avoid query N + 1 problem by using LEFT JOIN FETCH
@Query("SELECT s FROM Student s LEFT JOIN FETCH s.department LEFT JOIN FETCH s.courses")
List<Student> getAllStudents();

// Avoid query N + 1 problem by using EntityGraph
@EntityGraph(attributePaths = { "department", "courses" })
List<Student> findByStatus(StudentStatus status);

// Avoid query N + 1 problem by using EntityGraph
@EntityGraph(value = "Student.WithDepartmentAndCourses", type = EntityGraph.EntityGraphType.LOAD)
List<Student> findByDeleted(boolean deleted);
```

JPA Entity Graphs for Dynamic Fetching

```java
@PersistenceContext
private EntityManager em;

public List<StudentResponseDto> findByDepartmentId(Long departmentId) {
    // Using EntityManager to fetch students by department ID
    EntityGraph<?> graph = em.createEntityGraph(Student.class);
    graph.addAttributeNodes("department", "courses");
    List<Student> students = em
            .createQuery("SELECT s FROM Student s WHERE s.department.id = :departmentId", Student.class)
            .setHint("javax.persistence.loadgraph", graph) // Sử dụng Entity Graph
            .setParameter("departmentId", departmentId)
            .getResultList();

    // Convert to DTOs
    return students.stream()
            .map(this::convertToDto)
            .collect(Collectors.toList());
}
```

JPA Projections for Efficient Data Retrieval

JPA Projections for Efficient Data Retrieval

- **What are Projections?**:
  - Retrieve only specific fields from entities instead of the entire object.
  - Implemented via interfaces or DTO classes to reduce data transfer.
- **Types**:
  - Interface-based: Define an interface with getter methods for desired fields.
  - DTO-based: Use a custom class for complex projections.
- **Explanation**:
  - Returns only id, name, and email, reducing data overhead.
  - Works with Spring Data JPA's query derivation or custom @Query.

JPA Projections for Efficient Data Retrieval

```java
public interface StudentProjection {
    Long getId();
    String getName();
    String getEmail();
}

@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    List<StudentProjection> findByNameContaining(String name);
}
```

JPA Query Caching with Second-Level Cache

JPA Query Caching with Second-Level Cache

- **What is Query Caching?**:
  - Caches query results to avoid repeated database hits for frequently executed queries.
  - Uses Hibernate's second-level cache (L2 cache) with providers like EhCache.
- **Setup**:
  - Enable L2 cache in application.properties:
    - spring.jpa.properties.hibernate.cache.use_second_level_cache=true
    - spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory
- **Explanation**:
  - Caches query results and entities; subsequent calls retrieve from cache.
  - Requires cache provider (e.g., EhCache) dependency in pom.xml.

JPA Query Caching with Second-Level Cache

❖ Add @Cacheable to entities:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Student {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
}
```

JPA Query Caching with Second-Level Cache

❖ **Query Cache Example**:

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    @QueryHints(@QueryHint(name = "org.hibernate.cacheable", value = "true"))
    List<Student> findByName(String name);
}
```

JPA Lifecycle Events and Listeners

JPA Lifecycle Events and Listeners

- **What are Lifecycle Events?**:
  - JPA triggers events during entity lifecycle: creation, update, deletion, etc.
  - Annotations: @PrePersist, @PreUpdate, @PostLoad, @PostRemove, etc.
- **Use Case**:
  - Automatically set creation/modification timestamps or enforce business rules.
- **Explanation**:
- @PrePersist: Executes before saving a new entity.
- @PreUpdate: Executes before updating an existing entity.

```java
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;

    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
        updatedAt = LocalDateTime.now();
    }

    @PreUpdate
    public void preUpdate() {
        updatedAt = LocalDateTime.now();
    }
}
```

JPA Native Queries

JPA Native Queries

- **What are Native Queries?**:
  - Execute raw SQL queries when JPQL is insufficient for complex operations.
  - Map results to entities, DTOs, or scalar values.
- **Types**:
  - Entity-mapped: Return managed entities.
  - Scalar/DTO-mapped: Return custom objects or raw data.
- **Explanation**:
  - nativeQuery = true: Indicates raw SQL instead of JPQL.
  - Useful for database-specific features or complex joins.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    @Query(value = "...", nativeQuery = true)
    List<Object[]> findStudentAndDepartment(@Param("name") String name);
}
```

JPA Specifications for Dynamic Queries

JPA Specifications for Dynamic Queries

- **What are Specifications?**:
  - Spring Data JPA feature to build dynamic, reusable query criteria.
  - Uses Specification interface to define predicates for filtering.
- **Use Case**:
  - Implement flexible search filters (e.g., search students by name, email, or department).
- **Explanation**:
  - JpaSpecificationExecutor: Enables repository to use Specification.
  - Combine multiple specifications with and(), or() for complex filters.

JPA Specifications for Dynamic Queries

```java
import com.example.demo.entities.Student;
import org.springframework.data.jpa.domain.Specification;

public class StudentSpecifications {
    public static Specification<Student> hasName(String name) {
        return (root, query, cb) → cb.like(cb.lower(root.get("name")), "%" + name.toLowerCase() + "%");
    }
}
```

JPA Specifications for Dynamic Queries

```java
import com.example.demo.entities.Student;
import org.springframework.data.jpa.domain.Specification;

public class StudentSpecifications {
    public static Specification<Student> hasName(String name) {
        return (root, query, cb) -> cb.like(cb.lower(root.get("name")), "%" + name.toLowerCase() + "%");
    }
}
```

```java
// JPA Specifications for Dynamic Queries
public List<Student> findByName(String name) {
    return this.studentJpaRepository.findAll(StudentSpecifications.hasName(name));
}
```

JPA Bulk Operations

Introduction to JPA Bulk Operations

- **Implementation**:
  - Use @Query with @Modifying in repository methods.
  - JPQL for entity-based updates; native SQL for complex cases.
- **Explanation**:
  - @Modifying: Indicates the query modifies data (UPDATE/DELETE).
  - Returns number of affected rows.
  - Use @Transactional in service layer to ensure atomicity.

Introduction to JPA Bulk Operations

```java
@Repository
public interface StudentJpaRepository extends JpaRepository<Student, Long>, JpaSpecificationExecutor<Student> {
    @Query("SELECT s FROM Student s LEFT JOIN FETCH s.department")
    List<Student> getAllStudentsWithDepartment();

    @Modifying(clearAutomatically = true)
    @Query("UPDATE Student s SET s.status = :status WHERE s.department.id = :departmentId")
    int updateStudentStatus(@Param("status") String status, @Param("departmentId") Long departmentId);

    @Modifying
    @Query("DELETE FROM Student s WHERE s.status = :status")
    int deleteInactiveStudents(@Param("status") String status);
}

```

Introduction to JPA Bulk Operations

```java
@Repository
public interface StudentJpaRepository extends JpaRepository<Student, Long>, JpaSpecificationExecutor<Student> {
    @Query("SELECT s FROM Student s LEFT JOIN FETCH s.department")
    List<Student> getAllStudentsWithDepartment();

    @Modifying(clearAutomatically = true)
    @Query("UPDATE Student s SET s.status = :status WHERE s.department.id = :departmentId")
    int updateStudentStatus(@Param("status") String status, @Param("departmentId") Long departmentId);

    @Modifying
    @Query("DELETE FROM Student s WHERE s.status = :status")
    int deleteInactiveStudents(@Param("status") String status);
}

@Transactional
public int updateStudentStatus(Long deptId, String status) {
    return this.studentJpaRepository.updateStudentStatus(status, deptId);
}
```

Soft Delete in JPA

Introduction to Soft Delete in JPA

- **What is Soft Delete?**:
  - Marks records as "deleted" (e.g., via a flag) instead of physically removing them.
  - Preserves data for auditing, recovery, or compliance.
- **Why Use Soft Delete with Filters?**:
  - Automatically excludes "deleted" records from queries without modifying each query.
  - Implemented using Hibernate filters in JPA.
- **Key Components**:
  - A deleted flag in the entity (e.g., boolean or timestamp).
  - @Filter and @FilterDef annotations for global filtering.
- **Benefits**:
  - Simplifies queries; maintains data integrity.

Implementing Soft Delete with Filters

- **Steps**:
  - Add a deleted field to the entity.
  - Define @FilterDef and @Filter on the entity.
  - Enable the filter in queries via EntityManager.
- **Explanation**:
  - @FilterDef: Defines the filter with parameters.
  - @Filter: Applies the condition (e.g., deleted = false).
  - Enable filter per session for "active" records.

Implementing Soft Delete with Filters

```
@Transactional
public void softDeleteStudent(Long id) {
    Student student = this.studentJpaRepository.findById(id).orElseThrow();
    student.setDeleted(true);
    this.studentJpaRepository.save(student);
}
```

Implementing Soft Delete with Filters

```java
@FilterDef(name = "softDeleteFilter", parameters = @ParamDef(name = "deleted", type = Boolean.class))
@Filter(name = "softDeleteFilter", condition = "deleted = :deleted")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```java
public List<StudentResponseDto> findAvailableStudents() {
        Session session = em.unwrap(Session.class);
        session.enableFilter("softDeleteFilter").setParameter("deleted", false);
        List<Student> student = em
                .createQuery("SELECT s FROM Student s LEFT JOIN FETCH s.department LEFT JOIN FETCH s.courses",
                        Student.class)
                .getResultList();
        session.disableFilter("softDeleteFilter");
```

Soft Delete Operations and Best Practices

- **Soft Delete Operation**:
  - Set deleted = true instead of repository.delete().
- **Restoring Records**:
  - Set deleted = false to "undelete".
- **Best Practices**:
  - Use a timestamp for deletedAt for better auditing.
  - Enable filter globally via interceptor or aspect for consistency.
  - Combine with projections to exclude deleted field in responses.
- **Pitfalls**:
- Forgotten filter enablement leads to including "deleted" records.
- Performance impact on large datasets; index the deleted column.

JPA Custom Converters

Introduction to JPA Custom Converters

- **What are Custom Converters?**:
  - Map non-standard Java types to database columns.
  - Use @Converter to define custom conversion logic.
- **Use Cases**:
  - Enums to strings, JSON to text, custom objects to blobs.
  - Ensures type safety and portability.
- **Types**:
  - AttributeConverter: Implements AttributeConverter<X, Y> for entity attributes.
  - Auto-apply or explicit via @Convert.

Implementing JPA Custom Converters

❖ Create enum

```
@Getter
public enum StudentStatus {
    ACTIVE(code:"ACT"), INACTIVE(code:"INA"), SUSPENDED(code:"SUS");
    private final String code;
    StudentStatus(String code) { this.code = code; }

    public static StudentStatus fromCode(String code) {
        for (StudentStatus s : StudentStatus.values()) {
            if (s.code.equals(code)) return s;
        }
        throw new IllegalArgumentException("Invalid code: " + code);
    }
}
```

Implementing JPA Custom Converters

❖ Create Converter

```
// autoApply = true: This converter will be applied to all fields of type StudentStatus
// autoApply = false: This converter must be explicitly specified in the entity field
// If you want to use this converter for a specific field,
// you must annotate that field with @Convert(converter = StudentStatusConverter.class)
@Converter(autoApply = false)
public class StudentStatusConverter implements AttributeConverter<StudentStatus, String> {
    @Override
    public String convertToDatabaseColumn(StudentStatus status) {
        return status != null ? status.getCode() : null;
    }

    @Override
    public StudentStatus convertToEntityAttribute(String code) {
        return code != null ? StudentStatus.fromCode(code) : null;
    }
}
```

Implementing JPA Custom Converters

❖ Apply to entity

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

    @Column
    @Convert(converter = StudentStatusConverter.class)
    private StudentStatus status;
```

Conclusion and Next Steps

- **Summary**:
  - Learned to build a CRUD application with Spring Boot, JPA, and MySQL.
  - Covered entities, repositories, REST APIs, and basics JPA features.
- **Next Steps**:
  - Build a full-stack application with a front-end (e.g., React).
- **References**:
  - Spring Boot
  - Spring Data JPA
  - JPA Specification