**WEEK-3**

**SPRING DATA JPA AND HIBERNATE HANDSON**

**Objective 1: Demonstrate Implementation of Query Methods in Spring Data JPA**

Spring Data JPA enables the creation of query methods directly in repository interfaces by adhering to method naming patterns. These methods are interpreted by Spring at runtime, eliminating the need to write SQL or JPQL manually.

**Common Query Method Scenarios:**

1. **Search by Partial Match**

List<Employee> findByNameContaining(String keyword);

Fetches employees whose names include the specified keyword. (Case-sensitive by default.)

1. **Prefix Matching**

List<Employee> findByDepartmentStartingWith(String prefix);

Returns departments or names that start with the provided string.

1. **Ordering Results**

List<Employee> findByDepartmentOrderByNameAsc(String department);

Retrieves employees in a given department, ordered alphabetically by name.

1. **Querying Within a Date Range**

List<Employee> findByJoiningDateBetween(LocalDate start, LocalDate end);

Fetches employees who joined between the specified start and end dates (useful for monthly or weekly reports).

1. **Numeric Comparisons**

List<Employee> findBySalaryGreaterThan(Double minSalary);

List<Employee> findBySalaryLessThanEqual(Double maxSalary);

Filters employees based on salary thresholds.

1. **Top Records Retrieval**

List<Employee> findTop3ByOrderBySalaryDesc();

Fetches the top 3 employees with the highest salaries, useful in dashboards and analytics.

These methods can be combined and extended, offering a powerful way to handle queries with minimal boilerplate code.

**Objective 2: Demonstrate Implementation of Object-Relational Mapping (O/R Mapping)**

Spring Data JPA allows object-relational mapping using annotations, helping connect Java objects to relational tables. This is especially useful when working with multiple related entities like Employee, Department, and Project.

**Common Relationship Annotations:**

1. **@ManyToOne**

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "department\_id")

private Department department;

* Indicates that multiple employees belong to a single department.
* @JoinColumn sets the foreign key column.
* FetchType.LAZY ensures the department is only loaded when accessed.

1. **@OneToMany**

@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)

private List<Employee> employees;

* Defines that a department can have multiple employees.
* mappedBy points to the owning side in the Employee entity.

1. **@ManyToMany**

@ManyToMany

@JoinTable(

name = "employee\_project",

joinColumns = @JoinColumn(name = "employee\_id"),

inverseJoinColumns = @JoinColumn(name = "project\_id")

)

private Set<Project> projects;

* Represents a many-to-many relationship where employees can work on multiple projects and vice versa.
* @JoinTable defines the intermediate (junction) table with foreign keys.

1. **Fetch Strategies: LAZY vs EAGER**

* FetchType.LAZY: Data is fetched only when accessed—preferred for performance.
* FetchType.EAGER: Data is loaded immediately—can lead to overhead if overused.