**JPA (Java Persistence API) – Specification :**

* **JPA is a Java specification** for **Object-Relational Mapping (ORM)**, which allows Java objects to be persisted to relational databases.
* It is **just a set of interfaces and annotations** defined in the javax.persistence (or **jakarta.persistence** in newer versions) package.
* **JPA by itself does not perform any ORM logic** — it defines how ORM should be implemented.
* To actually persist and retrieve data, you need a **JPA provider/implementation** like:

**Hibernate** (most widely used), **EclipseLink**, **OpenJPA**

**Hibernate :**

* Hibernate is a popular **ORM framework** that **implements** the **JPA (Java Persistence API)** specification.
* Manages all the heavy lifting of ORM.
* Handles object-to-table mapping.
* Provides powerful features like: **Lazy loading, Transactional management, HQL (Hibernate Query Language), caching.**
* Hibernate = Actual Tool/Engine for ORM

Hibernate is a open source ORM framework in Java.

Inside hibernate there we need two files

* **HBM (Hibernate Mapping) File**: Defines the mapping between Java entity classes and database tables (e.g., Employee class to employee table using employee.hbm.xml).
* **CFG (Configuration) File**: The hibernate.cfg.xml file contains database-specific configurations like username, password, JDBC URL, and dialect.

To perform any database operation in Hibernate, we need a **Session** object.  
To obtain a Session, we first require a **SessionFactory**, which is typically created using a **HibernateUtil** class.  
Inside HibernateUtil, a **Configuration** object is used to load the configuration details from the hibernate.cfg.xml file.  
Once the Session object is available, we can use it to interact with the database using methods like session.save(), session.update(), session.delete(), or session.createQuery().

**Methods :**

get() and load() are methods used in Hibernate to retrieve entities by their primary key.

1. **get()**

immediately hits the database and returns the entity or null if not found. It returns the actual object and is eager.

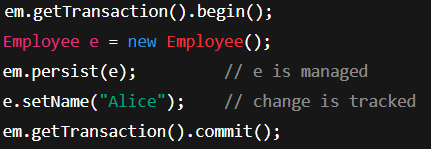
1. **load()**

returns a **proxy** at runtime and that proxy object **contain the logic** to hit to the database and when we call any getter methods like getName() it will hit the database immediately. The database is queried only when a property of the object is accessed **(lazy loading)** like getName(). If the entity does not exist, load() throws an **ObjectNotFoundException**.

1. **persist():**

**Manageable state -> changes are made -> tracked -> commit() -> if already present -> throw error**

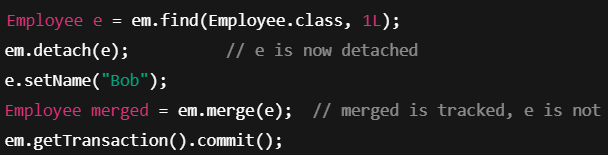
used to save new entity to the database. First entity is converted into manageable state, then changes are made which is tracked and then we use commit() operation to save the changes into the database and if the entity is already present then it will throw an error.



1. **merge() :**

when you want to add/ update an old/detached entity in the database. It **creates a copy** of the given entity, merges the changes into that copy, and returns the **managed version**. The original object remains **detached** and is **not tracked** by the persistence context.

**Create copy -> merge changes -> return merged version -> original remains as it is**

****

1. **beginTransaction()**
2. **getTransaction()**

**Caching in Hibernate (1st and 2nd level) :**

Caching is used to optimize performance by minimizing unnecessary database access. Hibernate provides two levels of caching: **First-Level Cache** and **Second-Level Cache**.



* The **First-Level Cache** (**session level cache**) is **built-in** and enabled by default. It is associated with the **Session object**, meaning that all entities fetched during a session are stored in this cache.
* If the same entity is requested again within that session, Hibernate retrieves it directly from memory instead of hitting the database again.
* This cache is **automatically managed and requires no configuration**.

**Second Level Cache:**



* the Second-Level Cache is an **optional**, **configurable cache** that works at the **SessionFactory level**, making it **shared across all sessions**.
* It is designed to avoid hitting the database for **frequently accessed data**, especially **read-mostly or reference data** — think of things like country lists, categories, or user roles.
* Hibernate stores entities in the cache after they are loaded for the first time from the database. On subsequent requests — even across different sessions — Hibernate first checks the second-level cache before querying the database. This greatly **reduces latency and improves application performance**.
* To enable second-level caching, you need to **configure it in the hibernate.cfg.xml (or application.properties in Spring Boot**) and integrate it with a **cache provider** like **EHCache**, **Caffeine**, **Redis** or **Hazelcast** (each offering different strategies and configurations to manage cached data effectively).
* For example, with EHCache, you add the dependency, enable caching using **hibernate.cache.use\_second\_level\_cache=true**, and specify the provider class **(hibernate.cache.region.factory\_class)**.
* You also create an ehcache.xml file to define cache regions and settings.
* We can qualify which classes or which entities qualify for the second level cache in the configuration or by using the annotation **@Cacheable.**

**EntityManager :**

EntityManager is an interface provided by Java Persistence API (JPA) that manages the lifecycle of entities, handles persistence operations, and interacts with the database.

It is part of Jakarta Persistence (formerly Java EE/JPA) and is commonly used in Spring Data JPA for advanced database operations.

Key Responsibilities of EntityManager

* CRUD Operations (Create, Read, Update, Delete)
* Managing Transactions
* Query Execution (JPQL, Native SQL)
* Caching and Persistence Context
* Entity Lifecycle Management (e.g., Merge, Detach, Remove)

**How to Get an EntityManager in Spring Boot?**

In Spring Boot + Spring Data JPA, EntityManager is usually injected via @PersistenceContext or retrieved from EntityManagerFactory.

**Hibernate with JDBC (Standalone):**

**Plain JDBC + Hibernate = manual setup and management.**

* You configure Hibernate manually using hibernate.cfg.xml or programmatic config.
* You manage SessionFactory and Session yourself.
* You write your own transaction management (begin, commit, rollback).
* Hibernate handles ORM: mapping Java objects to DB tables, generating SQL, managing entities.
* You write DAO classes to use Hibernate API directly.
* No built-in integration with other frameworks; everything is wired manually.

**Hibernate with Spring Boot:**

**Spring Boot + Hibernate = auto-configuration, simplified coding, and integrated transaction & dependency management.**

* Spring Boot auto-configures Hibernate via spring-boot-starter-data-jpa.
* EntityManagerFactory and SessionFactory are managed by Spring container.
* Transactions are managed declaratively using @Transactional.
* Data access is simplified via Spring Data JPA repositories — no boilerplate DAO code.
* Hibernate works seamlessly with Spring’s dependency injection, caching, and other features.
* Easier integration with other Spring modules like Security, MVC, and Cache.

**Spring Data JPA (Only for Relational Databases) :**

Spring Data JPA is a part of the Spring ecosystem that makes it super easy to use JPA with Spring Boot.

It sits on top of JPA + Hibernate and provides:

* Boilerplate-free repository interfaces
* **Automatic query generation** with the help of custom query methods.
* CRUD repository support
* Paging and sorting

**Spring Data JPA is built on top of ORM:**

Spring Data JPA uses **JPA (Java Persistence API)**, and JPA is an **ORM specification**.  
Under the hood, Spring Data JPA often uses **Hibernate**, which is a popular **ORM framework**.

**Query Methods:**

There are three main ways to implement custom methods, depending on what you need.

**1. Query Methods by Naming Convention**

* Spring Data JPA lets you define query methods by following naming conventions (e.g., **findByX, findByXBetween, findByXStartingWith**).
* It dynamically generates **SQL/JPQL** at runtime based on the method name and entity class.
* No manual query writing is needed for simple queries.
* These methods can be used in the service layer or directly in controllers.
* **Greatly reduces boilerplate code and simplifies repository interactions.**

**List<Person> findByLastName(String lastName);**

The logic behind each method is translated into SQL queries based on the entity class (Person) and method name.

**Custom Queries:**

**2. Using @Query Annotation (JPQL or Native SQL)**

* JPQL (Java Persistence Query Language)
* Native Queries (SQL specific to the database)

1. **JPQL (Java Persistence Query Language)**

**JPQL is object-oriented** and works with entity classes and their attributes instead of database tables.

**Advantages of JPQL**

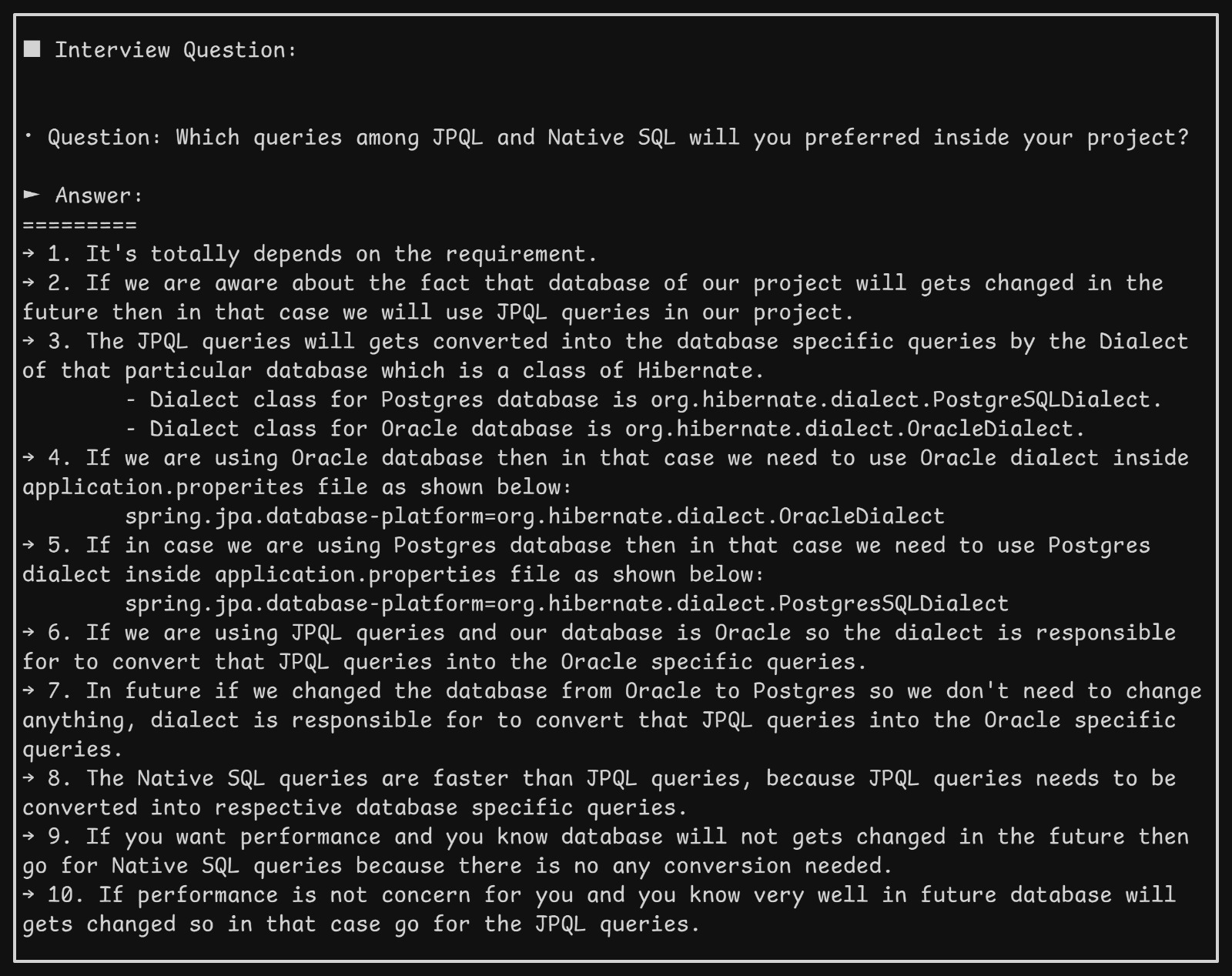
1. Works across multiple databases (database-agnostic).
2. Uses entity names and attributes, not table names and column names.
3. Supports **auto-mapping** between entities and query results.

**Disadvantages of JPQL**

1. Cannot use database-specific functions (e.g., JSONB in PostgreSQL).
2. **Slower** than native queries for complex operations.

**@Query("SELECT e FROM Employee e WHERE e.department = :dept")**

**List<Employee> findByDepartment(@Param("dept") String department);**

****

1. **Native Query (SQL Query specific to database)**

Native queries **use** **raw SQL** directly against the database.

**Advantages of Native Queries**

1. **Faster** for complex queries (e.g., joins, aggregations).
2. Supports database-specific functions (e.g., TO\_CHAR() in Oracle, JSONB in PostgreSQL).
3. Useful when JPQL does not support certain queries.

**Disadvantages of Native Queries**

1. Tied to a specific database (**not portable**).
2. Does not automatically map results to entities (unless specified).

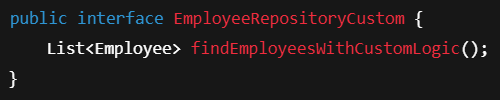
**@Query(value = "SELECT \* FROM employees WHERE department = :dept", nativeQuery = true)**

**List<Employee> findByDepartmentNative(@Param("dept") String department);**

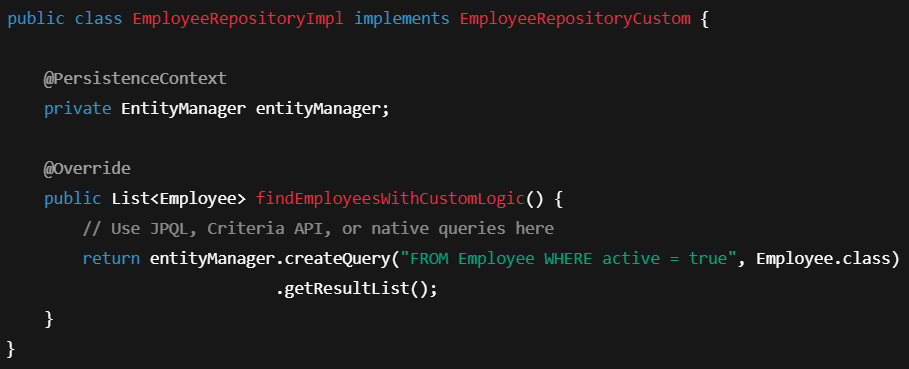
**3. Custom Repository Implementation (Advanced Logic)**

**If your logic is complex or involves multiple steps, use this method.**

**Step 1: Create a custom interface**

****

**Step 2: Implement the interface**

****

**Step 3: Extend the custom interface in your main repo**

**public interface EmployeeRepository extends JpaRepository<Employee, Long>, EmployeeRepositoryCustom {**

**}**

**What is ORM?**

**ORM (Object-Relational Mapping)** is a technique that allows you to map Java (or other language) objects to **relational database tables**.

ORM bridges the gap between **object-oriented programming** and **relational databases**.

**mapping** refers to associating one set of values with another. The types of mappings you use depend on the context — databases, programming (like Java, Python), or data modelling. Here's a breakdown of the most common types of mapping:

**1. One-to-One Mapping**

* **Definition**: One entity maps to exactly one other entity.
* **Example**: Each person has one unique PAN card.
* **Use Case**: User and Aadhaar, Employee and Passport.

**2. One-to-Many Mapping**

* **Definition**: One entity maps to multiple entities.
* **Example**: One PAN can have multiple financial records (one for each year).
* **Use Case**: Customer and Orders, Author and Books.

**3. Many-to-One Mapping**

* **Definition**: Many entities map to a single entity.
* **Example**: Many employees work in one department.
* **Use Case**: Salesman and Region.

**4. Many-to-Many Mapping**

* **Definition**: Many entities map to many entities.
* **Example**: Students enrolled in multiple courses, and each course has many students.
* **Use Case**: Tags on blog posts, Projects and Employees.

**5. Composite Key Mapping**

* **Definition**: A combination of two (or more) fields uniquely identifies a record.
* **Example**: (PAN, Financial Year) → Financial Details.
* **Use Case**: Transaction records with (UserID, Date), Attendance logs.

**6. Hierarchical Mapping**

* **Definition**: Parent-child relationship; like a tree.
* @EmbeddedId (for composite keys)
* **Example**: Categories and subcategories, Organization chart.
* **Use Case**: File directories, Menu navigation.

**7. Self-Mapping (Recursive)**

* **Definition**: An entity is mapped to itself.
* **Example**: An employee can be a manager of another employee.
* **Use Case**: Reporting hierarchy, Dependency graphs.

**CascadeType :**

**Cascading in Hibernate :**

1. **Cascading** in Hibernate means that **certain operations** (like save, delete, update) performed on a **parent entity** are automatically **propagated to its associated child entities**.
2. If I save/delete the parent, Hibernate will automatically save/delete the child too.
3. This helps reduce boilerplate code and ensures consistency in entity relationships.
4. Without cascading operations on child must be done manually but with cascading operations on parent will also propagate to child also.
5. Cascading works only for **relationships** (@OneToOne, @OneToMany, etc.)

There are various types of cascades in Hibernate:

* CascadeType.PERSIST Saves child entities when parent is saved
* CascadeType.MERGE Updates child entities when parent is merged
* **CascadeType.REMOVE Deletes child entities when parent is deleted**
* CascadeType.REFRESH Refreshes child entities when parent is refreshed
* CascadeType.DETACH Detaches child entities when parent is detached
* CascadeType.ALL Applies all of the above operations

**E.g. @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)**

**private List<Employee> employees = new ArrayList<>();**

**FetchType :**

**Fetch** defines **when** related entities should be loaded from the database:  
Immediately (EAGER) or only when needed (LAZY).

**EAGER** : Load the related entity immediately when the parent is loaded. It is better for large data or collection, Used default in @OneToMany, @ManyToMany

**LAZY** : Load the related entity only when accessed (on-demand). Can cause performance issues like over fetching. Used as default on @ManyToOne, @OneToOne.

**LazyInitializationException**

**Repositories in Spring Data JPA:**

* 1. **CrudRepository**

**CrudRepository** is the **base interface** for generic CRUD operations like -

* **C**reate – Add a new record (e.g., add a new user).
* **R**ead – Retrieve or view data (e.g., get all users).
* **U**pdate – Modify existing data (e.g., change a user's email).
* **D**elete – Remove data (e.g., delete a user).

Consists methods like :

**save(S entity)** Save or update an entity

**findById(int id)** Find entity by id

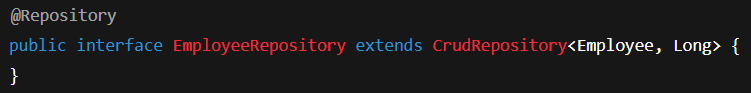
**existsById(int id)** Check if entity exists

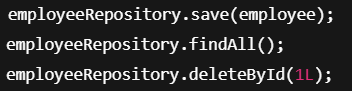
**findAll()** Find all entities

**deleteById(int id)** Delete by id

**count()**  Total number of records

It is **simple** and **sufficient for basic CRUD applications**.





* 1. **PagingAndSortingRepository**

It **extends CrudRepository** and adds pagination and sorting features.

**Pagination :**

Pagination is the process of dividing a large set of data (like a list of users, products, posts, etc.) into smaller chunks or pages, so you don’t load everything at once.

Improves performance, user experience and Saves memory and bandwidth.

Spring Data JPA provides a built-in way to handle pagination using Pageable and Page<T>

**Pageable :**

Pageable is an **interface** that **represents pagination information**.

It tells which page you want, how many items per page, and (optionally) how to start them.

Consists info like page number, page size, Sort information.

**PageRequest :**

PageRequest is an **implementation of Pageable**.

You usually create PageRequest objects when you want to paginate.



With sorting price by descending



**Sorting**

Sort is used to define sorting logic. You can sort by one or multiple columns, ascending or descending.

Sort sortByNameAsc = Sort.by("name").ascending();

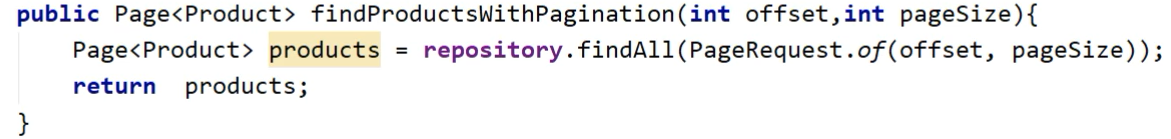
Sort sortBySalaryDesc = Sort.by("salary").descending();

Sort sortByMultipleFields = Sort.by("department").ascending().and(Sort.by("joiningDate").descending());

**Extra Methods:**

**findAll(Pageable pageable)** – Page<T> findAll(Pageable pageable);

* It **returns a Page object** — not just list of data but also metadata like total pages, total records, current page, etc.
* Very useful for making "Page 1", "Page 2", "Next", "Previous" buttons on web apps



**findAll(Sort sort)** – get sorted data

* Useful when your database has **lots of records** and you want **pages** or **sorted output**.
* Simple and fast when you don’t need paging, only ordering.

@Repository

public interface ProductRepository extends PagingAndSortingRepository<Product, Long> { }

Page<Product> page = productRepository.findAll(PageRequest.of(0, 5, Sort.by("price")));

Page 0, 5 products per page, Sorted by price.

**3. JpaRepository**

This is the most powerful repository. Full Power: **CRUD + Pagination + Batch + JPA Advanced**

* Extends PagingAndSortingRepository
* Adds batch operations, flush, and advanced JPA features.

Provides methods like:

saveAll(Iterable<S> entities) Save list of entities

flush() Push all pending changes to DB immediately

saveAndFlush(S entity) save and immediately flush

deleteInBatch(Iterable<T> entities) delete in batch

findAll(Pageable pageable) Pagination

findAll(Sort sort) Sorting

* Custom Repository interfaces
* Extending multiple repositories
* Optional<T> usage with findById

**Transaction**

A **transaction** is a logical unit of work in a **database** or **application** that consists of one or more operations (such as queries or updates) which are executed together. These operations must meet certain properties to ensure consistency and integrity. A transaction typically ensures that either **all** of the operations complete successfully (commit), or **none** are applied (rollback) in case of failure.

**Transaction Lifecycle:**

1. **Start**: The transaction begins.
2. **Execute Operations**: The transaction performs database operations (like inserts, updates, deletes).
3. **Commit**: If all operations succeed, the transaction is committed, making the changes permanent.
4. **Rollback**: If any operation fails, the transaction is rolled back, undoing all changes made during the transaction.

**Key Properties are ACID :**

1. **Atomicity :**

A transaction is atomic, meaning that all operations within the transaction are treated as a single unit.

If one operation fails, the entire transaction is rolled back, and no changes are saved to the database.

1. **Consistency:**

The database must always be in a consistent state before and after the transaction.

All business rules and constraints (foreign key , unique key constraints etc) must be satisfied.

1. **Isolation :**

Transactions are isolated from each other, meaning that the changes made by one transaction are not visible to the others until the transaction is committed.

Isolation ensures that transactions do not interfere with each other, even if they are running concurrently.

1. **Durability :**

Once a transaction is committed, the charges are permanent and will survive any system crashes or failures.

The changes are saved to the database and will not be lost.

**@Transactional annotation :**

**@Transactional** annotation in Spring is used to **manage transactions automatically**,

Ensures that the **entire method executes as a single transaction**.

If any exception is thrown, the transaction is **rolled back**.

Useful for **batch updates**, **chained database operations**, and **business logic consistency**.

**Rollback on Failure**

Spring **automatically rolls back** a transaction **if a RuntimeException or Error occurs** inside a **@Transactional** method**.**

**Rollback on : unchecked exception ( RuntimeException, Error )**

**Commit on : checked exceptions ( Exception subclass )**

**Use of @RequestParam**

@RequestParam is used in Spring MVC to **bind HTTP request parameters** (query parameters or form data) directly to method parameters in a controller.  
It allows the user to **send values through the URL**, and your controller method can easily access and use those values.

**Questions :**

**What is CRUD and JPA Repository?**

**Difference between Hibernate and Spring Data JPA?**

**Difference between @Entity, @Table and @Id:**

**@Entity:**

1. Marks a class as an **entity** that maps to a database table.
2. Required for any class to be managed by JPA/Hibernate.
3. By default, the table name is **the same as the class name**.

**@Table:**

1. Used to **customize the table name** for an entity.
2. If omitted, JPA will use the **class name** as the table name.

**@Id:**

1. Specifies **which field is the primary key** in the entity.
2. Required for every JPA entity.

**If I don’t want to save a particular field inside an entity in Hibernate**

if you don’t want to **persist** (save) a particular field in an entity (i.e., prevent it from being mapped to the database), you can use the **@Transient annotation**. This annotation tells Hibernate (or JPA) **not to map** the field to a database column, meaning the field will be ignored during persistence operations (like save, update, etc.) e.g. **@Transient**

private String temporaryField; // This will not be saved in the database

**How do you avoid LazyInitializationException?**

**LazyInitializationException :**

This exception occurs when a lazy-loaded entity is accessed **after the Hibernate session is closed**. For example, if you fetch an entity in a service method, but access its lazy collection in the controller **after the transaction ends**, it throws this exception.

To avoid **LazyInitializationException :**

* **Use @Transactional** this keeps the session open during the method that accessed the lazy data.
* Fetch with JPQL and JOIN FETCH – write a custom query to load the relation eagerly just for that case.
* Use DTO – fetch only required fields using a constructor-based DTO. etc

**How to configure JPA inside spring boot?**

Add dependencies (pom.xml for Maven)

Configure Database Connection (application.properties or application.yml) e.g. Database connection (database URL, Password, username, driver etc) JPA settings

* Create entity classes
* Create Repository Interfaces
* Use it in your Service / Controller

**@value and @Async**

**Explain how @Transactional works in Spring. What happens behind the scenes? Where to write @transactional?Method level or class level?**

**How does Spring Data JPA work under the hood?**

**What is Spring Data JPA? How is it different from Hibernate?**

**What is the purpose of JpaRepository?**

**What is the difference between CrudRepository, PagingAndSortingRepository, and JpaRepository?**

**How does Spring Data JPA implement repository methods without any code (like findAll(), save())?**

**What are derived query methods?  
(e.g., method names like findByUsername, findBySalaryGreaterThan)**

**What is the use of @Query annotation?**

**What is the difference between JPQL and Native SQL in Spring Data JPA?**

**What is the purpose of Projections in Spring Data JPA?**

**How do you perform pagination and sorting with Spring Data JPA?**

**Explain the use of @Transactional in repository methods.**

**How is lazy loading and eager loading handled in JPA?**

**What happens internally when you call save() method in a repository?**

When we call the save() method in a Spring Data JPA repository, it checks if the entity has an ID. If the ID is null, it treats it as a new entity and performs an INSERT. If the ID exists, it checks whether the entity is already in the database—if yes, it performs an UPDATE, otherwise an INSERT. Internally, it uses the EntityManager’s persist() or merge() methods. The changes are managed within a transaction and sent to the database when the transaction commits or the persistence context is flushed.

**Specifications (Dynamic Queries)**

**What are Specifications?**

**Specification** is a **design pattern** used for building **dynamic database queries**.

In Spring Data JPA**, Specification<T>** is a **functional Interface**.

**You use it to build queries at runtime based on user input.**

Instead of creating **many different repository methods**, you build flexible queries dynamically.

E.g. If I want to search **Employees** based on **optional parameters** like Department, minimum salary, name constraints keywords.

We cannot write 8-9 different methods(findByName, findByDepartment, findBySalary, findByNameAndDepartment, etc). Instead, we **dynamically** build query depending on which filter is provided.

Using Specification<T> interface

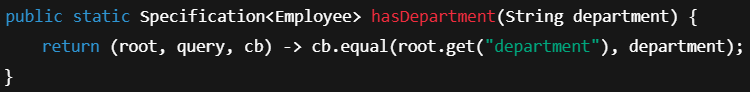
**@FunctionalInterface**

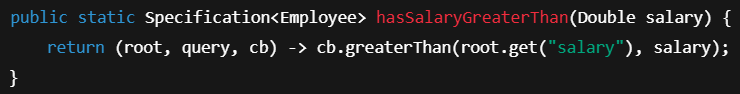
**public interface Specification<T> {**

**Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder);**

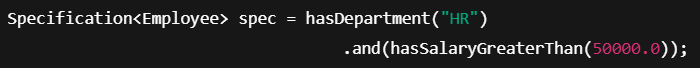
**}**

* Root<T> → access entity attributes (columns)
* CriteriaQuery<?> → represents the overall query
* CriteriaBuilder → helps build predicates (conditions)





Combine specification using .and(), .or(), .not().



This builds a query: **WHERE department = 'HR' AND salary > 50000**

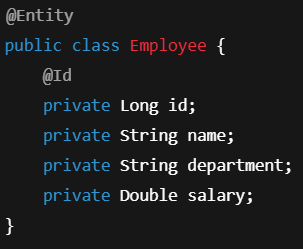
**Projections and DTOs**

1. **Interface-based projections**

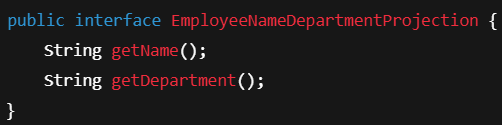
Here we define **Java Interface**.

Spring will **auto-populate fields** based on getter names.

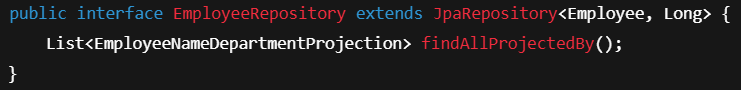
Step 1. Create Employee class



Step 2. Create Projection Interface.



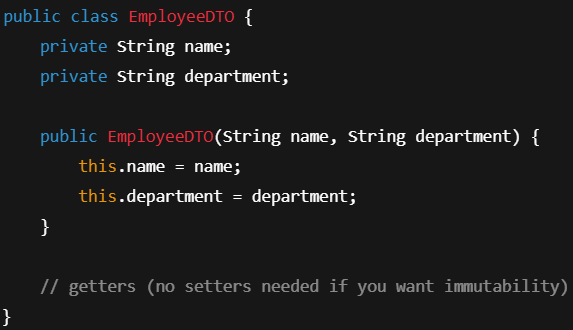
Step 3. Use in Repository.



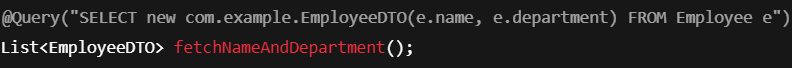
1. **Class-based projections (DTOs)**

Define a **simple POJO class (DTO).** Write **constructor** that matches the fields you want. Spring will fetch selected fields **and map them into the DTO**.

Step 1. Create a DTO



Step 2. Mention JPQL query to fetch required data according to DTO in Repository

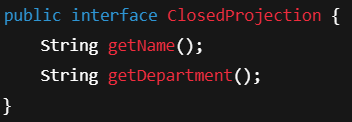


Here, you are using JPQL to **select only specific fields** and Spring auto maps them into your DTO.

1. **Open and closed projections**

**Closed Projection**

1. **Exact mapping** of DB columns to interface methods.
2. No custom methods or logic allowed.
3. Fetches **only fields you explicitly define**.



**Open Projection**

1. You can add **computed fields** inside your projection interface!
2. Spring will evaluate expressions using **SpEL (Spring Expression Language)**.
3. Not just fetching from database, but also **post-processing values**.

**Drawback** : Open projection might fetch entire entity first (then process it in memory) – not always efficient for very large queries.

