2.3 The Persistence Layer and JPA Repository

Persistence Layer (or Data Access Layer):

- The **Persistence Layer** is the part of a software application that is responsible for ***interacting*** with the ***database*** or any other ***persistent storage system*** to save, retrieve, update, or delete data.
- It acts as a ***bridge*** between the ***business logic*** (**service layer**) and the ***database***, abstracting away the **low-level database operations** (like SQL queries, connections) from the rest of the application.
- The ***Persistence Layer*** deals with ***Java Entity Objects***, not raw SQL or database tables.
- The ***ORM*** (like **Hibernate/JPA**) takes care of ***converting*** these **objects** into **actual database operations**.

ORM (Object-Relational Mapping):

ORM is a **programming technique** that allows developers to ***map*** (**connect**) ***Java Objects*** to ***Database Tables***, so that you can interact with your database using ***Java objects*** *instead* of writing ***raw SQL queries***.

Why ORM ?

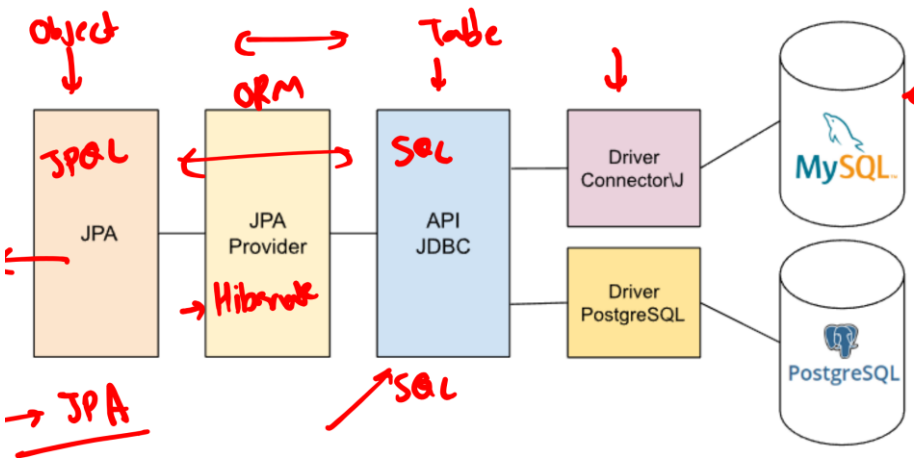
- **Databases** store data in ***tables*** (**rows** & **columns**).
- **Java** applications work with ***objects*** (**classes** & **attributes**).
- ORM ***bridges*** the gap between these two different paradigms.
- Reduces the need for ***manual SQL queries***, ***ResultSet parsing***, and ***boilerplate JDBC code***.
- **Popular ORM Frameworks in Java:**
    - **Hibernate** (Most widely used)
    - **EclipseLink**
    - **Spring Data JPA** (simplifies JPA + Hibernate usage)

JDBC vs Hibernate vs JPA:

Aspect	JDBC	JPA (Java Persistence API)	Hibernate
Type	Low-level API	Specification (Set of interfaces & rules)	Framework (JPA Implementation)
Purpose	*Direct interaction* with relational databases	Defines *how ORM* (Object-Relational Mapping) *should behave*	Provides *actual ORM functionality* (implements JPA)
Level of Abstraction	*Low* (Manual SQL, connection handling)	*High* (Object-oriented DB access model)	*High* (ORM abstraction over JDBC)
SQL Writing	You must *manually write SQL queries*	Uses *JPQL* (Java Persistence Query Language) or **derived queries**	Hibernate *auto-generates* SQL; can also use *HQL* (Hibernate Query Language)
Object Mapping	*Manual* ResultSet *mapping* to Java objects	Provides *annotations* to map *Java classes* to *DB tables*	Maps *Java objects* to *DB tables* using *annotations*/*XML*
Boilerplate Code	*High* (connection, statements, result sets, ...)	*Minimal* (Repositories, Annotations handle most of it)	*Minimal* (automates CRUD, caching, lazy loading)
Transactions	*Manual* transaction management	*Abstracted* transaction handling	Manages transactions *programmatically* or *declaratively*.
Usage	When fine-grained *SQL control* is needed	Preferred in *enterprise Java apps* for persistence	Most popular *ORM framework* in Java
Dependency	*No dependency* (part of core Java)	Needs a *JPA provider* like Hibernate.	Needs *Hibernate library*.
Relationship	Base API	Specification (interface)	Implementation of *JPA* (and can work without JPA too)

Note:

- JPA = Specification (Standard)
- Hibernate = ORM Tool (JPA Provider)
- JDBC = Low-level API for database interaction
- Driver = Translates JDBC calls to DB-specific protocol
- SQL is finally executed in DB.



What is JPA ?

- JPA is just a specification — it does not provide actual implementation.
- To use JPA, you need a *JPA Provider* (like *Hibernate*, EclipseLink, TopLink, etc.).
- JPA enables you to *persist* (save), *update*, *delete*, and *query* Java objects to/from relational databases.
- Uses *JPQL* (Java Persistence Query Language) — an *object-oriented* query language similar to SQL.
- It abstracts away the low-level *JDBC code* and provides a *cleaner*, *declarative* approach.

JPA migration:

To use *JPA* in Spring Boot maven project, you need to *add* the following *dependency* in your pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Additionally, we'll need a *Database Driver* dependency. (eg. MySQL, PostgreSQL, H2, ...)

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

@Entity (Java Persistence API)

Def^{n}: *@Entity* is a *JPA annotation* used to mark a *Java class* as a *persistent entity*, meaning it will be *mapped* to a *table* in the *database*.

Key Points:

- Represents a *table* in the *database*.
- Each *instance* of the class represents a *row*.
- Requires a field annotated with *@Id* (primary key).
- Works with *JPA*/*Hibernate* to store, retrieve, and manage data in relational databases.

```
@Entity
public class Employee {
   @Id   // now id is primary key of Employee table
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Long id;
   private String name;
}
```

@Repository (Spring Data)

Def^{n}: *@Repository* is a *Spring annotation* used to indicate that a *class* or *interface* is a *Data Access Object* (DAO), responsible for encapsulating the logic for accessing the *database*.

Key Points:

- Often used on *interfaces* that extend *JpaRepository*, *CrudRepository*, etc.
- Provides built-in *CRUD* operations without needing to write implementation.

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
   // No implementation needed
   /* extends JpaRepository<Employee, Long> -> means it generates SQL
      queries for employee class which primary key is Long type... */
}
```

Note: now object *EmployeeRepository* can call *.findById(id)*, *.findAll()*, *.save(Employee)*, *.saveAll(List<Employee>)* and many more without writing a single line of ddl or dql.

@Entity: Represents a table in your database — it's your *data model*.
@Repository: Provides CRUD operations (Create, Read, Update, Delete) on that *@Entity*.

Relation:

- The *@Repository* (typically through JpaRepository or CrudRepository) acts as a *bridge* between *@Entity* and the actual *database*.
- You use it to interact with the entity's table — for querying, saving, deleting records, etc.

So:

- *@Entity* defines *what* data is.
- *@Repository* defines *how* to access it (using Spring Data abstraction).

What is Lombok?

- *Lombok* is a Java library that helps you *reduce boilerplate code*. The repetitive code you write again and again — like *getters*, *setters*, *constructors*, *toString*(), and more.
- Instead of manually writing these methods, you can simply *use annotations*, and Lombok will generate them at *compile time*.

Annotation	What it Does
@Getter / @Setter	Generates getters and setters for all fields
@NoArgsConstructor	Creates a *no-arguments constructor*
@AllArgsConstructor	Creates a *constructor* with *all fields*
@RequiredArgsConstructor	Creates *constructo*r for *final/non-null fields*
@Data	*Combines @Getter*, *@Setter*, *@EqualsAndHashCode*, *@ToString*, *@RequiredArgsConstructor*
@Builder	Implements the *Builder pattern* automatically