=================
Spring Data JPA
=================

-> Spring Data JPA is one module in spring framework

-> It is used to develop Persistence Layer (DB Logic)

1) Java JDBC
2) Spring JDBC
3) Hibernate Framework
4) Spring ORM
5) Spring Data JPA

USER_MASTER ===> UserMasterDao ==> 4 methods (insert/update/delete/select)

ROLE_MASTER ===> RoleMasterDao ==> 4 methods (insert/update/delete/select)

PRODUCT_DTLS ===> ProductDtlsDao ==> 4 methods (insert/update/delete/select)

PAYMENT_DTLS ===> PaymentDtlsDao ==> 4 methods (insert/update/delete/select)

..
..

5000 tables ====> 5000 * 4 = 20,000 methods

Note: If we have 5000 DB tables then we have to create 5000 DAO classes. Every dao class should contain 4 common methods so it will become 20,000 methods with same logic. (This is not recommended).

=> To avoid boiler plate code in DAO classes we can use Spring Data JPA.

=> Spring Data JPA will use Hibernate framework internally.

============================
Spring Data JPA Repositories
============================

=> To simplify Persistence Layer Development Data JPA provided Repository interfaces

1) CrudRepository (CRUD Ops methods)

2) JpaRepository (CRUD Ops methods + sorting + pagination + QBE)

Note: To perform DB operations we need to create an interface by extending properties from JPA repository interface.

==
Developing First Data JPA Application
==

1) Create Spring Boot application with dependencies

a) data-jpa-starter
b) mysql-driver
c) project-lombok

2) Configure Datasource properites in application.properties file

3) Create Entity class (Java class to DB Table mapping)

4) Create Repository interface by extending JPA Repository interface

5) Test the application by calling Repository interface methods.

```java
@Data
@Entity
public class Book {

@Id
private Integer bookId;
private String bookName;
private Double bookPrice;
}
```

```java
public interface BookRepository extends CrudRepository<Book, Integer>{

}
```

```properties
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.url=jdbc:mysql://localhost:3306/sbms
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

```java
@SpringBootApplication
public class Application {

public static void main(String[] args) {
ConfigurableApplicationContext ctxt =
SpringApplication.run(Application.class, args);

BookRepository repo = ctxt.getBean(BookRepository.class);
/*
* Book b = new Book(); b.setBookId(101); b.setBookName("Spring");
* b.setBookPrice(1000.00);
*
* repo.save(b);
*
* System.out.println("Record inserted.....");
*/


Optional<Book> findById = repo.findById(101);
System.out.println(findById.get());

}
}
```

======================
CrudRepository Methods
=======================

1) save(E) : To insert & Update (Insert + Update = Upsert)

2) saveAll(Iterable) : To insert & update collection of records

3) boolean existsById(ID) : To check presense of record (true/false)

4) count ( ) : To get records count in table

5) findById(ID) : To retrieve record based on given PK

6) findAllById(Iterable ids): Retrieve records based on given PKs

7) findAll () : To retrieve all records from table

8) deleteById(ID) : Delete record based on given PK

9) deleteAllById (Iterable ids): Delete records based on given PKs

10) delete(E) : Delete record based on given Entity obj

11) deleteAll(Iterable entities): Delete records based on given Entity objs

12) deleteAll ( ) : Delete all records from table (truncate)

----

1) findBy Methods

2) Custom Queries

=================
Find By Methods

=================

-> findBy Methods are used to perform only select operations

-> Using Non Primary Key columns also we can select records

-> In findBy Methods method name is very important because based on method name JPA will construct the query for execution.

Note: findBy methods should represent Entity class variables

```
public interface BookRepository extends CrudRepository<Book, Integer> {

// select * from book where book_price > : price
public List<Book> findByBookPriceGreaterThan(Double price);

// select * from book where book_price < : price
public List<Book> findByBookPriceLessThan(Double price);

// select * from book where book_name = : bookName
public List<Book> findByBookName(String bookName);
}
```

===============
Custom Queries
===============

-> If we want to execute our query in JPA Repo then we can go for Custom Queries

-> Custom Queries we can write in 2 ways

1) HQL Queries
2) Native SQL queries

Note: To represent Custom Query we are using @Query annotation.

```
--------------------------------------------------------------------------
public interface BookRepository extends CrudRepository<Book, Integer> {

@Query(value = "select * from book", nativeQuery = true)
public List<Book> getAllBooks();

@Query("from Book")
public List<Book> getBooks();

}
```

=====================
HQL Vs SQL Queries
======================

#1
-> HQL queries are DB independent queries

-> SQL queries are DB dependent queries

#2
-> IN HQL query, we will use entity class name & variables
-> IN SQL query, we will use table name & column names

#3
-> HQL query can't execute in DB directley (conversion required)
-> SQL query can execute in DB directley

#4
-> Performance wise SQL queries are better
-> Maintanence wise HQL queries are better

Note: Every HQL query should be converted to SQL query before execution. That conversion will done by Dialect class.

-> Every Database will have its own Dialect class.

Ex:

OracleDialect
MySqlDialect
DB2Dialect
PostgresDialect etc...

Note: Dialect class will be loaded along with DB driver class.

SQL : Select * from book;
HQL : from Book;

SQL : select * from book where book_price=2000;
HQL : from Book where bookPrice = 2000;

SQL : select * from book where book_price >= 2000 and book_name='Spring'
HQL : From Book where bookPrice >= 200 and bookName = 'Spring'

SQL : select book_name from book;
HQL : select bookName from Book;

===============
JpaRepository
===============

-> It is a predefined interface available in data jpa

-> Using JpaRepository also we can perform CRUD Operations

-> We can perform Crud Ops + Sorting + Pagination + QBE

```
EmployeeRepository repository = context.getBean(EmployeeRepository.class);

Employee e1 = new Employee(2, "Orlen", 5000.00, "Male", "Sales");
Employee e2 = new Employee(3, "Charles", 15000.00, "Male", "Admin");
Employee e3 = new Employee(4, "Smith", 25000.00, "Male", "Marketing");
Employee e4 = new Employee(5, "Cathy", 35000.00, "Fe-Male", "Account");
Employee e5 = new Employee(6, "Robert", 45000.00, "Male", "HR");
Employee e6 = new Employee(7, "David", 55000.00, "Male", "Manager");

repository.saveAll(Arrays.asList(e1, e2, e3, e4, e5, e6));

System.out.println("record saved...");
```

----
findAll ( ) ;

findAll(Sort sort) ;

findAll (Pagebale pageable) ;

===========
Pagination
===========

=> Dividing total records into multple pages is called as Pagination

Total records : 7
Page_Size : 3

what is page Number : ?

1st Page : first 3 records

2nd page : 4,5,6 records

3 rd page : last record (7th record)

```
@SpringBootApplication
public class Application {

public static void main(String[] args) {
ConfigurableApplicationContext context =
SpringApplication.run(Application.class, args);

EmployeeRepository repository = context.getBean(EmployeeRepository.class);
Sort sort = Sort.by("empName", "empSalary").descending();
// List<Employe> emps = repository.findAll(sort);

int pageNo = 3;
PageRequest page = PageRequest.of(pageNo-1, 3);

Page<Employee> findAll = repository.findAll(page);
List<Employee> emps = findAll.getContent();

emps.forEach(System.out::println);
}
}
```

===================
Query By Example
===================

-> It is used to prepare Dynamic Query based on data available in Entity class obj

```
@SpringBootApplication
public class Application {

public static void main(String[] args) {
ConfigurableApplicationContext context =
SpringApplication.run(Application.class, args);

EmployeeRepository repository = context.getBean(EmployeeRepository.class);

Employee emp = new Employee();


Example<Employee> exmp = Example.of(emp);
List<Employee> emps = repository.findAll(exmp);

emps.forEach(System.out::println);
}
}
```

===========
Assignment
===========

1) Update record in table using custom query

2) Delete record in table using custom query

3) Insert record in table using custom query

=====================
TimeStamping In JPA

=====================

-> It is used to insert CREATED_DATE & UPDATED_DATE columns values for the record

@CreationTimeStamp : To populate record created date

@UpdateTimeStamp : To populate record updated date

```
@Entity
@Table(name = "emp_tbl")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Employee {

@Id
private Integer empId;
private String empName;
private Double empSalary;
private String empGender;
private String dept;

@CreationTimestamp
@Column(name="date_created", updatable = false)
private LocalDate dateCreated;

@UpdateTimestamp
@Column(name="last_updated", insertable = false)
private LocalDate lastUpdated;

}
```

============
Generators
============

-> Generators are used to generate the value for Primary key Columns

-> Primary Key is a constraint in database it is used to maintain unique data in column

-> Primary Key constraint is combination of below 2 constraints

1) UNIQUE
2) NOT NULL

We can specify below strategies for Generator

1) AUTO
2) IDENTITY (mysql)
3) SEQUENCE (oracle)

4) TABLE : It will maintain another table to get primary key column values

===================
Custom Generator
===================

=> We want to generate primary key columns values as per project requirement like below.

ORDER_DTLS table

OD_1
OD_2
OD_3
...

OD_100

-> To achieve this requirement we need to develop our own generator. That is called as Custom Generator.

```
public class OrderIdGenerator implements IdentifierGenerator {

public void generate(){
String prefix = "OD_"
String suffix = get from db; [1,2,3...]

String pk = prefix+suffix;

//set pk to entity obj
}
}
```

Custom Generator : https://youtu.be/IijGVtT9ZPk

===

```
// to create sequence in oracle database
create sequence order_id_seq
start with 1
increment by 1;

// To get next value from sequence
select order_id_seq.nextval from dual;
```

=======================
Composite Primary Keys

======================

=> More than one Primary Key column in table

=> Below is sql query to create a table with multiple primary key columns

```
create table account_tbl (
acc_num bigint not null,
acc_type varchar(255) not null,
branch varchar(255),
holder_name varchar(255),
primary key (acc_num, acc_type)
) ;
```

-> To work with Composite PKs we will use below 2 annotations

@Embeddable : Class level annotation to represent primary key columns

@EmbeddedId : Variable level annotation to present Embdedable class

Note: Generators will not support for Composite Primary keys

Note: The class which is representing Composite Keys should implement Serializable interface.

```
-------------------------------------------------------------------------------
@Data
@Embeddable
public class AccountPK implements Serializable{

private Long accNum;

private String accType;

}
-------------------------------------------------------------------------------
@Data
@Entity
@Table(name="account_tbl")
public class Account {

private String holderName;

private String branch;

@EmbeddedId
private AccountPK accountPk;

}
-------------------------------------------------------------------------------
public interface AccountRepo
```

```
extends JpaRepository<Account, AccountPK>{
}
```

```
@SpringBootApplication
public class Application {

public static void main(String[] args) {
ConfigurableApplicationContext context =
SpringApplication.run(Application.class, args);

AccountRepo bean = context.getBean(AccountRepo.class);

/*

// setting composite key values
AccountPK pk = new AccountPK();
pk.setAccNum(15454323212l);
pk.setAccType("Savings");

// setting entity data
Account acc = new Account();
acc.setHolderName("Ashok");
acc.setBranch("Ameerpet");
acc.setAccountPk(pk); // setting pk obj

bean.save(acc); // saving the entity obj

System.out.println("Record saved.....");

*/

AccountPK pk = new AccountPK();
pk.setAccNum(15454323212l);
pk.setAccType("Savings");

Optional<Account> findById = bean.findById(pk);
if(findById.isPresent()) {
System.out.println(findById.get());
}

}

}
```

==

=======================
Profiles in Spring Boot
=======================

-> Every Project will have multiple Environments in the Realtime.

-> Environments are used to test our application before delivering to client

Note: Environment means platform to run our application

(Linux VM, DB Server, Log Server etc...)

-> We can see below Environments in the Realtime

1) Dev Env : Developers will use it for code Integration testing

2) SIT Env : Testers will use it application end to end testing

3) UAT Env : Client / Client Side Team will use it for acceptance testing

Note: In UAT client / client side team will decide GO or NO-GO

GO : Approved for Production deployment
NO-GO : Denied for Production Deployment

4) PILOT Env : Pre-Prod Env for testing with live data

5) PROD Env : Live Environment (endusers will access our prod app)

=> Every Environment will have its own Database and every database will have seperate configuration properties (uname, pwd and URL)

=> If we want to deploy our code into multiple envrionments then we have to change configuration properties in application.properties file for every deployment. (It is not recommended).

1) DB Props

2) SMTP PRops

3) Kafka Props

4) Redis Props

5) REST API Endpoint URLs

=> To avoid this problem we will use Profiles in Spring Boot.

=> Profiles are used to configure Environment Specific properties

application-dev.properties
application-sit.properties
application-uat.properties
application-pilot.properties
application-prod.properties

=> In application.properties file we need to activate profile

```
# Activating dev profile
spring.profiles.active=dev
```

Note: Above represents application should load properties from Dev properties file.