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

JPA : Java Persistence API

=> JPA is used to communicate with Databases by using Java applications.

=> We Have several options to communicate with database using java applications

        1) JDBC API (boiler plate code)

        2) Spring JDBC (we have to write queries, data will be represented in text format)

        3) Hibernate (ORM -> Object Relational Mapping)

        4) Spring ORM (Internally uses Hibernate) == 2 lines 2 lines  2 lines  2 lines

        5) Spring Data JPA (latest) => 1 line

=========================
What is Spring Data JPA ?
=========================

=> Spring Data JPA is part of the Spring Data project.

=> Spring Data JPA is used to simplify Database Operations.

=> It reduces boilerplate code required to implement data access logic

### Note: By Writing just 1 line of code we can perform CRUD operations in database table. ###

=> Spring Data JPA internally uses JPA specification. Hibernate is an implementation for JPA.

Note: Hibernate internally uses JDBC to execute SQL queries with database.


        Spring Data JPA --> JPA --> Hiberante --> JDBC --> Database

================================
Advantages with Spring Data JPA
================================

1) Automatic Repository Implementation : If we create Repository interface then Data JPA will
implement that in Runtime.

2) Query Methods : Allows you to define queries just by method names.

                public List<Emp> findByEmpGender(String gender);

3) JPQL and Native Queries : You can use @Query annotation to write custom JPQL or native SQL
queries.

4) Pagination and Sorting : Built-in support via Pageable and Sort parameters.

5) Transaction Management : Integrates smoothly with Spring's transaction management.

================================
Core Concepts of Spring Data JPA
================================

        1) Data Source
```

2) Entity

3) Repository Interfaces

==========
What is Datasource ?
==========

=> DataSource represents set of connections with database. It will create connection pool.

Note: We need to configure datasource properties in "application.properties" file.

```
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
```

==========
What is Entity ?
==========

=> Entity is a java class which is mapped with database table. We will use below annotations in entity class.

@Entity : Represents java class as Entity class (mandatory).

@Table : map class name with table name (optional).

@Id: Represents the variable mapped with PK column (mandatory).

@Column : Map entity class variable with db table column name (optional).

==========
What is Repository ?
==========

=> Repository is an interface provided by Data JPA to perform CRUD operations.

Note: In data jpa mainley we have 2 repository interfaces

1) CrudRepository : We can perform CRUD Operations

2) JpaRepository : CRUD Ops + Sorting + Pagination + Query By Example (QBE)

Ex:

```
public interface EmpRepository extends CrudRepository<Employee, Integer> {

}
```

==========
Environment Setup
==========

1) Database Setup
- MySQL DB Server
- MySQL workbench (Client s/w)

@@ MySQL DB Installation Video : https://www.youtube.com/watch?v=EsAIXPIsyQg

@@ Oracle DB Installation Video : https://www.youtube.com/watch?v=-RrYpn-ACAk

==========
Developing First Data JPA Application
==========

@@ Project Lombok Video : https://www.youtube.com/watch?v=8tDym-FxU0A

@@ Data JPA App development Video : https://www.youtube.com/watch?v=ZGKHCJsp4hg

1) Create SpringBoot application with below dependencies

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

2) Configure datasource & ORM properties in application.properties file

3) Create Entity class (table mapping)

4) Create Our Repository interface by extending CrudRepository interface.

5) Create Service class and inject Repository and write required methods

6) Test app by calling service class methods from start class.

==========
CrudRepository interface methods
==========

1) save (T obj) : INSERT or UPDATE single record ==> (UPSERT)

2) saveAll (List<T> objs) : INSERT or UPDATE list of records ==> (UPSERT)

3) findById(Id id) : Retrieve record using Primary Column Value

4) findAllById(List<ID> ids) : Retrieve records using multiple Primary Column Values

5) findAll( ) : Retrieve all records from table

6) exitsById(ID id) : Verify record is present in table or not.

7) count ( ) : To get count of total records present in table.

8) deleteById (ID id) : Delete record based on given Primary Key value

9) deleteAllById (List<ID> ids) : Delete list of records based on given Primary Key values

10) delete(T obj) : Delete record based on given entity obj

11) deleteAll(List<T> objs) : Delete records based on given entities

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

==========
Working with findByXXX() methods in data jpa
==========

=> To retrieve table data based on primary column value(s) we have predefined methods in data jpa like

1) findById (ID id),
1) findAllById(List<ID> ids)

####### Q) How to retrive records based on non-primary column values ? ########

=> Based on project requirement we need to retrieve records based on non-primary column values also like below

Ex : select * form user where gender='Male'

select * from user where country = 'India'

select * from user where country ='India' and gender='Fe-Male';

=> To achieve above requirement we can use findByXXX() methods in Data JPA.

=> findBy methods are used only to retrieve records from table (will not support for DML).

=> When we are working with findBy methods method name is very important because based on method name only JPA will construct query and will execute that query.

Note: in findBy method names we should use entity class variable names.

Note: findByXXX( ) methods we will write in Repository interface as abstract methods (no body). The methods implementation will be taken care by Data JPA in runtime.

```
@Entity
@Data
public class User {

	@Id
	private Integer uid;
	private String uname;
	private Integer age;
	private String gender;
	private String country;

}
```

```
public interface UserRepo extends CrudRepository<User, Integer> {

	// select * from user where gender=:male;
	public List<User> findByGender(String gender);

	// select * from user where country=:country;
	public List<User> findByCountry(String country);

	// select * from user where gender='Male' and country='India';
	public List<User> findByGenderAndCountry(String gender, String country);

	// select * from user where age >= 20;
	public List<User> findByAgeGreaterThanEqual(Integer age);

}
```

==========

Working with Custom Queries in data jpa

==========

=> We can write our own queries and we can ask Data JPA to execute our queries that is called as Custom Query.

=> We will write custom queries in Repository interface.

=> To execute custom queries we will use "@Query" annotation.

=> Custom queries we can write in 2 ways

1) Native SQL

2) HQL / JPQL

=> SQL stands for structured query language.
=> HQL stands for Hibernate query language.

=> SQL queries are database dependent.
=> HQL queries are database independent.

=> When we change application from one db to another db then we should change SQL queries also and we need to re-test entire application. This takes lot of time. Maintenence is difficult.

=> If we write HQL queries then those HQL queries will be converted into SQL queries based on Database configured in the application using Dialect classes.

Note : Dialect classes are used to convert HQL to SQL.

=> For every database we have dedicated Dialect class.

Oracle db ===> OracleDialect

MySQL db ===> MySqlDialect

DB2 DB ===> DB2Dialect

Note: Performance wise SQL is better, maintanence wise HQL is better.

=> In SQL query we will use table names and column names directley.

Ex: select * from user_info where user_country="India";

=> In HQL queries we will use entity class name and entity cls variable names.

Ex: From User where country="India";

SQL : select * from user_tbl
HQL : From User

SQL : select * from user_tbl where user_id=101
HQL : From User where userId=101

SQL : select * from user_tbl where gender='Male' and country='India'
HQL : From User where gender='Male' and country='India'

SQL : select user_id, user_name from user_tbl
HQL : select userId, userName from User

SQL : delete from user_tbl where user_id=101
HQL : delete from User where userId = 101

==========
Assignment : Perform INSERT, Update and DELETE operations using Custom queries.
==========

DML ===> INSERT + UPDATE + DELETE ==> TX required

=> JPA will take care of tx when we use pre-defined methods like save(), delete()...

=> If we want to execute custom query for "insert / update / delete" then we should use below 2

annoations

@Transacational

@Modifying

DQL ===> SELECT ===> TX Not Required

```
@Query(value = "delete from User where uid = :id")
@Transactional
@Modifying
public void deleteUserByHQL(Integer id);
```

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

=> This is a predefined data jpa interface which is used to perform DB operations.

=> Using JPA Repository we can perform below operations

1) Crud Operations

2) Sorting

3) Pagination

4) Query By Example

JpaRepository = CrudRepository + Pagination + Sorting + QBE

==========
What is Sorting ?
==========

=> It is used to retrieve records in ascending or descending order

Note: When we use sort in data jpa, it will add order by clause to the query

---

```
public void getUsersWithSorting() {

        Sort sort = Sort.by("age").ascending(); // order by age

        List<User> all = userRepo.findAll(sort); // select * from user order by age

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

==========
What is Pagination ?
==========

=> The process dividing records into multiple pages.

Ex: If we have 1 lakh records in table it is not recommended to display one lakh records in single page.

=> In realtime we need to implement pagination in our application.

```
                flipkart --> will display 24 records in one page

                gmail ---> will display 50 records in inbox

-------------------------------------------------------------

public void getUsersByPageNum(int pageNum) {

        /*
         * List<User> usersList = userRepo.findAll();
         * usersList.forEach(System.out::println);
         */

        // represents pagination
        PageRequest page = PageRequest.of(pageNum - 1, pageSize);

        Page<User> pageData = userRepo.findAll(page);

        List<User> usersData = pageData.getContent();
        // Stream<User> stream = pageData.get();

        usersData.forEach(System.out::println);
}
-------------------------------------------------------------

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

=> It is used to construct dynamic query based on filter conditions.

=> Based on entity obj data query will be created with conditions to filter records from table.

-------------------------------------------------------------
public void getUsersWithQBE() {

        User user = new User();
        // user.setGender("Male");
        // user.setCountry("India");
        // user.setAge(22);

        Example<User> example = Example.of(user);

        List<User> all = userRepo.findAll(example);

        all.forEach(System.out::println);
}
-------------------------------------------------------------

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

Requirement : Develop data jpa application to retrieve users based on below conditions with dynamic
query

Condition : gender=Male and age >= 35


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

=> When we are working with ORM framework for DB Operations, primary key is mandatory in every DB
```

table.

=> Primary Key is a constraint which is used to maintain unique records in the table.

=> Primary key is a combination of 2 contraints

@ Primary Key = NOT NULL + UNIQUE

=> While inserting records in table, We shouldn't set primary key column values to entity manually in the program.

=> Generators are used to generate values for primary key columns.

=> In Data JPA we have below 5 strategies for generators

A) AUTO (depends on DB)

B) IDENTITY (MySQL -> AUTO_INCREMENT)

C) SEQUENCE (Oracle)

D) TABLE (new table for primary keys)

E) UUID (alphanumeric) (Ex: 446b9b23-748e-4556-9978-520e3f4c6314)

==========
What is Custom Generator ?
==========

=> Custom Generators are used to generate primary key column value based on our project requirement.

Requirement-1 : Generate primary key column (emp_id) values like below for inserting employees into table.

Ex:

AIT1
AIT2
AIT3
..

AIT250

Requirement-2 : Generate primary key column (order_id) values like below for inserting orders into table.

Ex:

OD1
OD2
OD3
OD4
...

Note: To implement above 2 requirements we can't use predefined generators.

=> To generate primary key value according to client given requirement we should create our own Generator class which is called as Custom Generator.

=> To create our own generator, we need to implement one interface i.e "IdentifierGenerator" and override generate() method.

Note: Inside generate() method we should logic according to our requirement.

@@ Custom Generator Example Video : https://youtu.be/IijGVtT9ZPk?si=gyjATE7nMgllX_kH

1) What is Spring Data JPA

2) Advantages with Spring data JPA

3) Spring Data JPA Architecture

4) App development using Spring Data JPA

5) CrudRepository methods

6) findByXXX methods

7) Custom queries (SQL vs HQL)

8) Generators

```
====================================
How to work with In memory Database
====================================
```

=> Oracle & MySQL Databases are called as External databases. We need to download and install these databases in our system.

=> Data stored in External Databases is permanent. Data remains perminently even if our application is not running.

=> Spring Boot supports In Memory databases also like H2.

=> In Memory DB means when app starts DB will come and when app stopped then DB will be removed.

Note: In Memory DBs are used for practice purpose and for
POC (Proof of concept) development.

------------------ working with In-Memory database ---------------

Step-1 : Create boot app with below dependencies

1) data-jpa-starter
2) h2 driver
3) web-starter
4) project lombok

Step-2 : Configure datasource props in "application.properties" file

```
# Data source properties
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=ashokit
spring.datasource.password=abc@123

spring.h2.console.enabled=true
```

Step-3 : Create Entity class & repo interface

```
-----------------------------------------------------
@Entity
@Data
```

```
public class Product {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Integer productId;

        private String productName;

        private Double productPrice;

        @CreationTimestamp
        @Column(updatable = false)
        private LocalDate createdDate;

        @UpdateTimestamp
        @Column(insertable = false)
        private LocalDate updatedDate;

}
```

Step-4 : Service class with business methods

Step-5 : Test application methods

==========

Spring Boot with AWS RDS : https://www.youtube.com/watch?v=GSu1g9jvFhY

==========

1) Spring Data JPA with Oracle

2) Spring Data JPA with MySQL

3) Spring Data JPA with H2 (only for practice)

4) Spring Data JPA with AWS RDS

5) Spring Data with Mongo DB Integration

==========

Q) How to connect with multiple databases using springboot

insert emp record in MYSQL DB

insert product record in H2 DB

Referene Video : https://www.youtube.com/watch?v=mIFIb_JE47U

==========

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

=> SpringBoot Profiles are used to maintain environment specific configurations.

=> In real-time we will have multiple environments to test our application.

1) DEV

2) SIT
3) UAT
4) PILOT
5) PROD (live)

=> DEV env used by developers to perform Integration testing.

=> SIT env used by testers to perform system integration testing.

=> UAT env used by client side team to perform user acceptance testing.

=> PILOT env used for pre-prod testing.

=> PROD env used for live deployment. End users will access the application which is running in PROD Env.

Note: ENV to ENV some properties will be different

Ex: datasource, smtp, payment gateway, kafka ...

=> When we are deploying application to paritcular environment we need to make changes to application.properties file.

=> Everytime changing properties file is not recommended. It is time taking process and there is a chance of doing mistakes also.

Note: To avoid these problems we will use Spring Boot Profiles concept.

=> Using profiles, we will create seperate properties for every env like below..

application-dev.properties

application-sit.properties

application-uat.properties

application-pilot.properties

application-prod.properties

=> From which file it should load the properties that we are going to configure in main "application.properties" file. This is called as activating profile.

spring.profiles.active=dev

==========
Association Mapping
==========

=> For every application multiple database tables will be available. Those Tables will have relationships.

=> One table data will have relationship with another table data like below...

Ex-1 : Employee and Address (one emp can have mutliple addresses)

Ex-2 : Country and State (one country having multiple states)

Ex-3 : Book and Author (one author can publish multiple books)

Ex-4 : User and Role (one user can have multiple roles)

Note: To establish relationships among db tables we will use FOREGIN KEY concept.

=> Association mapping is used to represent DB tables relationships in Entity classes.

```
=> Relationships we can divide into 4 types

                1) One To One

                2) One To Many

                3) Many To One

                4) Many to Many


####  1) One To One #####

        Ex: Person & Passport

Note: One record in parent table will have relationship with one record in child table.

                Ex: One person will have only one passport.

Note: Here we can store person_id as a foreign_key in passport_table (or) we can store passport_id as
foreign_key in person_tbl to represent relationship.

=> To represent one to one relationship among entities we will use @OneToOne annotation.

=> To represent foregin key we will use @JoinColumn annotation.

Note: in OneToOne relation default FetchType is EAGER.

-----------------------------------------------------------------
public void savePersonWithPassport() {

                Passport pp = new Passport();
                pp.setPassportNum("J97979SDF");
                pp.setIssuedDt(LocalDate.now());
                pp.setExpDt(LocalDate.now().plusYears(10));

                Person p = new Person();
                p.setName("John");
                p.setGender("Male");

                // associating entities
                p.setPassport(pp);
                pp.setPerson(p);

                // save the person
                personRepo.save(p);

                System.out.println("Record saved...!!");
        }
-----------------------------------------------------------

#### 2) One To Many #####

        Ex : Employee with addresses
        Ex : Author with Books
        Ex : Trainer with Courses
        Ex : Category with Products
        Ex : University with Colleges

Note: One record in parent table will have relationship with multiple records in child table.

        Ex: One employee will have multiple addresses (present & permanent)

Note: We should store emp_id as foregin key in address table.
```

=> To represent one to many relationship we will use @OneToMany annotation.

Note: In OneToMany relation default FETCH TYPE is LAZY. If we want to load child records also along with parent record then we can set Fetch Type as EAGER like below...

```
@OneToMany(mappedBy = "emp", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
private List<Address> addrList;
```

```
public void saveEmpWithAddr() {

                Address a1 = new Address();
                a1.setCity("Hyd");
                a1.setState("TG");
                a1.setType("PRESENT");

                Address a2 = new Address();
                a2.setCity("Guntur");
                a2.setState("AP");
                a2.setType("PERMENENT");

                Employee e = new Employee();
                e.setName("Raj");
                e.setSalary(45000.00);

                // associating emp obj with addr objs
                a1.setEmp(e);
                a2.setEmp(e);

                // associate addrs objs with emp obj
                e.setAddrList(Arrays.asList(a1, a2));

                // save parent record
                empRepo.save(e);

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

        }
```

---

3) Many To One

Ex: Multiple books with single author

Note : Many records in one table will have relationship with single record in another table.

=> To represent Many To One relationship we will use @ManyToOne annotation.

4) Many To Many

Ex: Users with Roles

Note: Multiple users will have relationship with multiple roles.

user_tbl : users data will be stored

role_tbl : roles data will be stored

user_roles : join table

=> When DB tables having relationships then we have to represent those relationships in Entity classes which is called as Association Mapping.

=> To establish assocation mapping in entity classes we will use below annotations...

```
@OneToOne
@OneToMany
@ManyToOne
@ManyToMany

@JoinColumn
@JoinTable
```

==========

mappedBy : It is used to represent mapped variable of child class in parent entity.

cascade : It is used to represent which operations on parent table should reflect in child table.

fetchType : EAGER & LAZY

EAGER LOADING : when parent record is loaded, then by default load child records also.

LAZY loading : When parent record is loaded then don't load child records by default.

Note: in OneToOne relation default FetchType is EAGER.

Note : in OneToMany relation default FetchType is LAZY.

==========