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

=> Application contains several Layers

1) Presentation Layer (JSP / Thymeleaf / Angular / React JS / Vue JS)

2) Web Layer (Servlets / Struts / Spring Web MVC)

3) Persistence Layer (JDBC / Spring JDBC / Spring ORM / Spring Data JPA)

=> Spring Data JPA is used to develop Persistence layer in the application.

=> Spring Data JPA providing ready made methods to perform CRUD operation in DB tables.

=> Data JPA providing ready made methods using interfaces like below

1) CrudRepository (I)

2) JpaRepository (I)

Note : JpaRepository = CrudRepository + Pagination Methods + Sorting Methods

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

1) Data Source Object : It represents Database Connections

Note: Data Source properties we can configure in "application.properties" or "application.yml" file

2) Entity Class : The class which is mapped with database table

```
@Entity
@Table
@Id
@Column
```

3) Repository interface : For every Table we will create one repository interface to perform Crud Operations

```
public interface StudentRepository extends CrudRepository<Student, Integer>{

}
```

Note: For our Repository interface, implementation will be provided in the runtime using Proxy Class.

Note: By using StudentRepository we can perform CRUD operations in STUDENT_TBL

4) Repository methods : Ready made methods provided by Data JPA to perform CRUD operations

1) save (Entity)

2) saveAll (Iterable<Entity> i)

Note: Above two methods are called as "UPSERT" methods (UPDATE + INSERT)

```
3) findById (ID id)

4) findAllById (Iterable<ID> ids)

5) findAll ( )

6) count ( )

7) existById (ID id)

8) deleteById (ID id)

9) deleteAllById (Iterable<ID> ids)

10) deleteAll ( )


####### 5) ORM Properties : To automate some configurations

1)  auto_ddl : Dynamic Schema Generation

2) show_sql : Display generated queries on the console



==============================================
First Application Development Using Spring Data JPA
==============================================

1) Create Spring Starter Project with below dependencies

                a) springboot-starter-data-jpa

                b) mysql-driver

2) Create entity class and map with DB table using annotations

3) Create Repository interface to perform CRUD operations

4) Configure Data Source properties in application.yml file

5) Run the application and test the functionality


======================Student.java=======================
@Entity
@Table(name = "STUDENT_DTLS") // optional
public class Student {

        @Id
        @Column(name="student_id") // optional
        private Integer id;

        @Column(name="student_name")
        private String name;

        @Column(name="student_rank")
        private Long rank;

        @Column(name="student_gender")
        private String gender;

        //setters & getters
```

```
}

==========================StudentRepository.java==================
package in.ashokit.repository;

import org.springframework.data.repository.CrudRepository;

import in.ashokit.entity.Student;

//@Repository
public interface StudentRepository extends CrudRepository<Student, Integer>{

}

============================application.yml file====================
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: AshokIT@123
    url: jdbc:mysql://localhost:3306/sbms27
    username: ashokit
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

==================================Start Class============================

package in.ashokit;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

import in.ashokit.entity.Student;
import in.ashokit.repository.StudentRepository;

@SpringBootApplication
public class Application {

        public static void main(String[] args) {

                ConfigurableApplicationContext context = SpringApplication.run(Application.class,
args);

                StudentRepository studentRepo = context.getBean(StudentRepository.class);

                Student s = new Student();
                s.setId(103);
                s.setName("Ashok");
                s.setRank(1133l);
                s.setGender("Male");

                studentRepo.save(s);

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


                Student s1 = new Student(103, "Orlen", 90l, "Male");
                Student s2 = new Student(104, "Cathy", 100l, "FeMale");
                Student s3 = new Student(105, "Buttler", 70l, "Male");
                Student s4 = new Student(106, "Smitha", 60l, "FeMale");

                List<Student> students = Arrays.asList(s1, s2, s3, s4);
```

```
                studentRepo.saveAll(students);

                Optional<Student> optional = studentRepo.findById(101);

                if(optional.isPresent()) {
                        System.out.println(optional.get());
                }


                Iterable<Student> findAll = studentRepo.findAll();

                // findAll.forEach(System.out::println);

                findAll.forEach(s -> System.out.println(s));


                if(studentRepo.existsById(503)) {
                        studentRepo.deleteById(503);
                        System.out.println("Record Deleted....");
                }else {
                        System.out.println("No Record Present...");
                }

        }
}
```

```
============================
findByXXX methods in Data JPA
============================

=> By using findByXXX ( ) methods we can retrieve the data based on  non - primary key columns also

=> When we write findByXXX method , JPA will construct query based on method name

Note: Method Naming convention is very important for findByXXX methods

=> Using findBy methods we can perform select operations only (retrieval).   INSERT / UPDATE / DELETE
operations we can't do using findBy methods.

Note : In findBy method syntax we will use entity variable names.



====================================Entity Class====================================
```

```
@Entity
@Table(name = "STUDENT_DTLS") // optional
public class Student {

        @Id
        @Column(name = "student_id") // optional
        private Integer id;

        @Column(name = "student_name")
        private String name;

        @Column(name = "student_rank")
        private Long rank;

        @Column(name = "student_gender")
        private String gender;

        // setters & getters

}
```

```
==========================Repository Interface==========================

public interface StudentRepository extends CrudRepository<Student, Integer>{

        // select * from student_dtls where student_gender=:gender
        public List<Student> findByGender(String gender);

        // select * from student_dtls where student_gender is null
        public List<Student> findByGenderIsNull();

        // select * from student_dtls where student_rank >= : rank
        public List<Student> findByRankGreaterThanEqual(Long rank);

        // select * from student_dtls where student_rank <= : rank
        public List<Student> findByRankLessThanEqual(Long rank);

        // male students who are having rank >=100 ;
        // select * from student_dtls where student_gender=? and student_rank >= :rank

        public List<Student> findByGenderAndRankGreaterThanEqual(String gender, Long rank);

}


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

=> We can execute Custom Queries also in JPA (our own queries)

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

=> @Query will support for executing both HQL queries & Native SQL queries also.


        ###  HQL : Hibernate Query Language ( Database Independent Queries )

                        => In HQL, we will use Entity class name & Entity class variables to write
query

                        => HQL queries will converted to SQL queries by Dialect class for execution

                        => If we change app from one DB to another DB then no need to change any
query
                             because Dialect class will take care of query conversion

                        => HQL queries will give poor performance because of conversion ( HQL -> SQL
)


         ###  SQL : Structured Query Language ( Database Dependent Queries )

                        => In SQL, we will use table name & column names to write the query

                        => SQL queries will directley execute in database

                        => If we change app from one DB to another DB then all queries may not
execute

                        => SQL queries will give better performance than HQL


============================ Repository with Custom Queries ======================
```

```
public interface StudentRepository extends CrudRepository<Student, Integer> {

        @Query(value = "select * from student_dtls", nativeQuery = true)
        public List<Student> getAllStudents();

        @Query("from Student")
        public List<Student> getStudents();

}
```

==========

1) Using JPA pre-defined methods (Select + Non - Select)

2) Using findByXXX methods   (Note: Only for select operations)

3) Using Custom Queries (Select + Non - Select)

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

```
SQL :   select * from student_dtls where student_gender=:gender
HQL :   from Student where gender=:gender

SQl :  select * from student_dtls where student_gender is null
HQL : from Student where gender is null

SQL : select * from student_dtls where student_rank >= : rank
HQL : from Student where rank >= :rank

SQL :  select * from student_dtls where student_rank <= : rank
HQL : from Student where rank <= :rank

SQL :  select * from student_dtls where student_gender=:gender and student_rank >= :rank
HQL : from Student where gender = :gender and rank >= :rank

SQL : select  student_rank, student_gender from student_dtls
HQL : select rank, gender from Student
```

==========

Selection : Retrieving specific rows from the table. We can achieve this by using ' where ' keyword in the query

Ex : select * from student_dtls where gender = 'Male' ;

Projection : Retrieving specific columns from the table is called as Projection.
We can achieve by using column names in query.

Ex: select student_rank, student_gender from student_dtls

Note: We can combine selection & projection in single query.

Ex: select student_rank, student_gender from student_dtls where student_rank <= 100 ;

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

=> It is predefined interface provided by Spring Data JPA

=> JpaRepository provided several methods to perform CRUD operations with database.

=> JpaRepository provided few additional methods to perform DB operations

```
                JpaRepository = CrudRepository + PagingAndSorting + QueryByExample



==================================================================
Pagination : Displaying table records in multiple pages is called as Pagination
==================================================================
Ex:

=> Google Search Results will display with pagination ( Page size : 10 )

=> Gmail inbox mails will display with pagination ( Page size : 50 )

=> Flipkart products will display with pagination ( Page size  : 24 )


================================Pagination Example================================

public class Application {

        public static void main(String[] args) {

                ConfigurableApplicationContext context = SpringApplication.run(Application.class,
args);

                StudentRepository studentRepo = context.getBean(StudentRepository.class);

                Scanner s = new Scanner(System.in);
                System.out.print("Enter Page Number");

                int pageNo = s.nextInt();
                int pageSize = 3;

                // Page Num will start from 0
                PageRequest pageReq = PageRequest.of(pageNo - 1, pageSize);
                Page<Student> page = studentRepo.findAll(pageReq);
                List<Student> students = page.getContent();
                students.forEach(System.out::println);

        }
}

============================ Sorting Example ================================

@SpringBootApplication
public class Application {

        public static void main(String[] args) {

                ConfigurableApplicationContext context = SpringApplication.run(Application.class,
args);

                StudentRepository studentRepo = context.getBean(StudentRepository.class);

                List<Student> students = studentRepo.findAll(Sort.by("name").descending());
                students.forEach(System.out::println);

        }
}


================
QueryByExample
================
```

=> QBE is used to construct select query dynamically based on given entity object data

```
@SpringBootApplication
public class Application {

	public static void main(String[] args) {

		ConfigurableApplicationContext context = SpringApplication.run(Application.class,
args);

		StudentRepository studentRepo = context.getBean(StudentRepository.class);

		Student s = new Student();
		s.setGender("Male");
		s.setRank(90l);

		Example<Student> example = Example.of(s);
		List<Student> students = studentRepo.findAll(example);
		students.forEach(System.out::println);

	}
}
```

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

=> Insert, Update & Delete records using Custom Queries in Data JPA

=> If we want to perform Non Select Operations using Data JPA custom query then we should use below 2 annotations at our method.

1) @Modifying

2) @Transactional

Note: The above annotations are not required for selection operation.

==========Example==========

```
public interface StudentRepository extends JpaRepository<Student, Integer> {

	@Query("delete from Student where id = :sid")
	@Modifying
	@Transactional
	public void deleteStudent(Integer sid);

	@Query("update Student set gender=:gender where id=:sid")
	@Modifying
	@Transactional
	public Integer updateStudent(Integer sid, String gender);

	@Query(value = "insert into student_dtls(student_id,student_name,student_gender) values(:id,
:name, :gender)", nativeQuery = true)
	@Modifying
	@Transactional
	public void insertStudent(Integer id, String name, String gender);

	@Query("from Student")
	public void selectStudents();

}
```

```
======================
Timestamping in Data JPA
======================

=> @CreationTimestamp : It is used to populate record inserted date into DB column

=> @UpdateTimestamp  : It is used to populate record updated date into DB column



@Entity
@Table(name = "STUDENT_DTLS") // optional
public class Student {

	public Student() {
	}

	public Student(Integer id, String name, Long rank, String gender) {
		this.id = id;
		this.name = name;
		this.rank = rank;
		this.gender = gender;
	}

	@Id
	@Column(name = "student_id") // optional
	private Integer id;

	@Column(name = "student_name")
	private String name;

	@Column(name = "student_rank")
	private Long rank;

	@Column(name = "student_gender")
	private String gender;

	@CreationTimestamp
	@Column(name = "CREATED_DATE", updatable = false)
	private LocalDateTime createDate;

	@UpdateTimestamp
	@Column(name = "UPDATED_DATE", insertable = false)
	private LocalDateTime updateDate;


		//setters & getters

}


Note: LocalDate class represents will date value, where as LocalDateTime class will represent Date
with Time.

======================
Soft Delete & Hard Delete
======================

=> Hard Delete means deleting the record from DB permanently using "Delete" query. Once we perform
Hard Delete we can't get data back from DB.

=> Soft delete means updating the record as IN-ACTIVE. Soft Deleted records will present in DB so we
can access whenever we want.
```

```
        Ex : Active / De-Activate

Note: We can implement SOFT DELETE using additional column in DB table (ACTIVE_SW)


                        ACTIVE_SW =>>>> Y ====> Active record

                        ACTIVE_SW =>>>> N ====> Deleted record


======================== Soft Delete Example ==================

@SpringBootApplication
public class Application {

        public static void main(String[] args) {

                ConfigurableApplicationContext context = SpringApplication.run(Application.class,
args);

                StudentRepository studentRepo = context.getBean(StudentRepository.class);

                Student student = studentRepo.findById(101).get();
                student.setActiveSW("N");

                studentRepo.save(student);


        }
}

====================================
Primary key & Composite Primary Key In DB
====================================

=> Primary key is a constraint to maintain unique records in the table

        Primary key = UNIQUE + NOT NULL

create table product (
        product_id integer not null,
        product_name varchar(255),
        product_price double precision not null,
        primary key (product_id)
);


=> If we try to insert duplicate value in PK column value then we will get UniqueConstraintException.

=> We shouldn't ask end users to enter value for PK column because users may enter duplicate value.

=> To generate value for PK column we will use GENERATOR concept.



======   MYSQL DB  ========

AUTO / SEQUENCE / TABLE ====> New table will be created to maintain primary column values

IDENTITY ===> Will use AUTO_INCREMENT to generate value for Primary key column value

Note: MySQL DB will not support Sequences.


======= Oracle DB ==========
```

```
AUTO  => sequence to generate primary key value (default sequence name : hibernate_sequence)

SEQUENCE ===> We can configure our own sequence to generate PK value like below

create sequence  pid_seq
start with 1000
increment by 1;

TABLE   ====> New table will be created to maintain primary column values

Note: Orace DB will not support for AUTO_INCREMENT



============================== Configuring Table Generator ==================

        @Id
        @TableGenerator(initialValue = 100, name = "pid", table="pid_seq_tbl")
        @GeneratedValue(strategy = GenerationType.TABLE, generator="pid")
        private Integer productId;


======================== Configuring Custom Sequence To Generate PK Value ==================

=> First we need to create sequence in db like below


create sequence  pid_seq
start with 1000
increment by 1;

=> Configure Custom Sequence in Entity class like below

        @Id
        @SequenceGenerator(name = "pid", sequenceName = "pid_seq")
        @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "pid")
        private Integer productId;


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

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

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


====================
Composite Primary Key
====================

-> If a table contains more than one PK column then it is called Composite Primary key


create table product (
        product_id integer not null,
        product_name varchar(255),
        product_price double precision not null,
        primary key (product_id, product_name)
);

-> When we have Composite PKs then the combination PK columns data shouldn't be repeated in table.

Note: We can't use generators to generate value for Composite Primary keys.
```

```
============================== Entity Class ============================

@Embeddable
public class AccountPK implements Serializable{

	private Integer accId;
	private String accType;
	private Long accNum;

	// setters & getters

}

@Entity
public class Account {

	private String holderName;
	private String branch;

	@EmbeddedId
	private AccountPK accountPk;

	//setters & getters

}

==============================Repository=======================================

public interface AccountRepository extends JpaRepository<Account, AccountPK> {

}

==============================Boot start class=====================================

@SpringBootApplication
public class Application {

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

		AccountRepository accountRepo = context.getBean(AccountRepository.class);

		/*AccountPK pk = new AccountPK();
		pk.setAccId(2);
		pk.setAccType("Current");
		pk.setAccNum(324356681);

		Account acc = new Account();
		acc.setHolderName("Raju");
		acc.setBranch("Ameerpet");
		acc.setAccountPk(pk);

		accountRepo.save(acc);*/

		AccountPK pk = new AccountPK();
		pk.setAccId(2);
		pk.setAccType("Current");
		pk.setAccNum(324356681);

		Optional<Account> findById = accountRepo.findById(pk);
		System.out.println(findById.get());
	}
}
```

```
============================
Connection Pooling in Data JPA
============================

=> Connection Pooling is used to maintain connections required for our application.

=> Getting Connection from DB directley is not recommended because it will degrade our application.

=> We need to create connection pool to store connections when our application starts.

=> To perform DB operations we need to get connection from Connection Pool instead of getting from
DB.

=> Using Connection Pool we can improve performance of the application.

Note: Spring Data JPA will use Hikari Connection Pool by default.



============================
Working with Stored Procedures
============================

=> Procedure means set of sql queries

=> Procedures are used to write business logic at database side

=> Procedures are used to improve performance of the application.


--------------------------- Procedure to retrieve data from Product table --------------------------
---------

DELIMITER $$
CREATE PROCEDURE getProducts()
BEGIN
    SELECT * FROM PRODUCT;
END$$
DELIMITER ;

call getProducts();

------------------------------------ Calling Procedure using JpaRepository ------------------------
--

public interface ProductRepository extends JpaRepository<Product, Integer>{

        @Query(value="call getProducts()", nativeQuery=true)
        public List<Product> getAllProducts();

}

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

1) What is Spring Data JPA ?
2) Hibernate Vs Data JPA ?
3) Spring Data JPA Repositories
4) What is Entity
5) Annotations to map Java class to DB table
6) How to develop application using Data JPA
7) CrudRepository methods
```

8) JpaRepository methods
9) CrudRepository vs JpaRepository
10) What is Pagination ?
11) Sorting
12) Query By Example
13) Working with findByXXX methods
14) Custom Queries using @Query (@Modifying, @Transactional)
15) HQL Vs SQL
16) What is Dialect ?
17) Calling Stored Procedures
18) Timestamping
19) Soft Delete Vs Hard Delete
20) Primary key
21) Generators
22) Composite Primary key (@Embeddeble, @EmbeddedId)
23) Connection Pooling (Hikari CP)
24) Spring Boot Profiles
25) How to develop Custom Generator

==========
Association Mapping / Relationships in DB tables
==========

=> We can divide DB side relations into 4 types

Note: To establish relationships between table we will use Foregin Key.

1) One To One (Ex: One Person will have one Passport)

2) One To Many (Ex : One Employee will have Multiple Addresses)

3) Many To One (Ex: Multiple Books belongs to one Author)

4) Many To Many (Ex: Multiple Users having Multiple Roles)

=> When DB tables are having Relation then we need to represent that relation in our Entity classes also.

=> The process of representing DB tables relation in Entity classes is called as Association Mapping.

Cascade Type : Default Type is NONE : It represents operations on parent record should reflect on child record or not

Ex: When we delete parent record then we want to delete all Child Records of that Parent.

Fetch Type : Default Type is LAZY : It represents weather load child records along with Parent or not

Ex: When we retrieve Parent record i want to retrieve all child records of that Parent.

==========One To Many Mappinng ==========

```
@Entity
@Data
public class Employee {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Integer empId;
	private String empName;
```

```
	private Double empSalary;

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

}

@Entity
@Data
public class Address {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Integer addrId;
	private String city;
	private String state;
	private String country;

	@ManyToOne
	@JoinColumn(name = "emp_id")
	private Employee emp;
}

public interface EmpRepository extends JpaRepository<Employee, Integer>{

}

public interface AddressRepository extends JpaRepository<Address, Integer>{

}

@SpringBootApplication
public class Application {

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

		EmpRepository empRepository = context.getBean(EmpRepository.class);
		AddressRepository addrReposiotry = context.getBean(AddressRepository.class);

		Employee e = new Employee();
		e.setEmpName("Raja");
		e.setEmpSalary(4000.00);

		Address a1 = new Address();
		a1.setCity("Hyd");
		a1.setState("TG");
		a1.setCountry("India");
		a1.setEmp(e);

		Address a2 = new Address();
		a2.setCity("GNT");
		a2.setState("AP");
		a2.setCountry("India");
		a2.setEmp(e);

		// setting addresses to emp
		List<Address> addrList = Arrays.asList(a1, a2);
		e.setAddr(addrList);

		// empRepository.save(e);

		// empRepository.findById(2);
```

```
                // empRepository.deleteById(1);

                // addrReposiotry.findById(3);
        }
}
```

==========