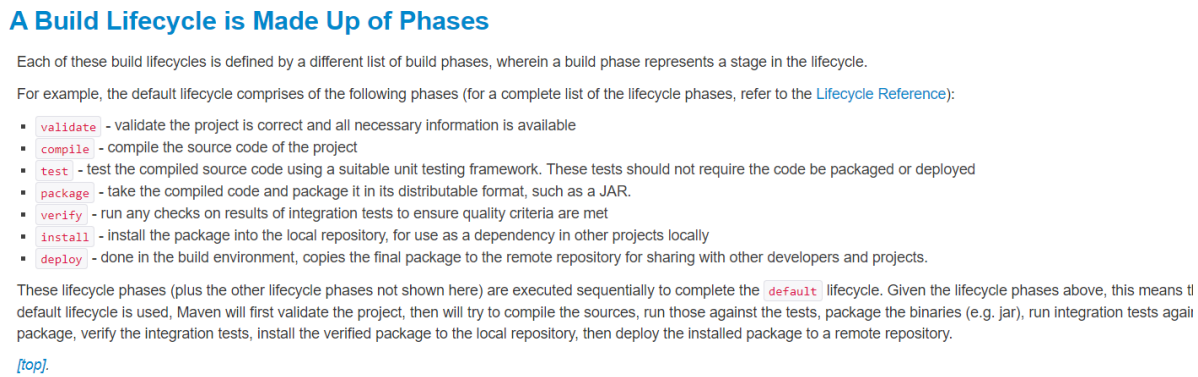
YT : https://youtu.be/6r-MpAWVw6c?si=fxsbHFsfNEql5n8F



A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

- `validate` - validate the project is correct and all necessary information is available
- `compile` - compile the source code of the project
- `test` - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package` - take the compiled code and package it in its distributable format, such as a JAR.
- `verify` - run any checks on results of integration tests to ensure quality criteria are met
- `install` - install the package into the local repository, for use as a dependency in other projects locally
- `deploy` - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the `default` lifecycle. Given the lifecycle phases above, this means tl default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests agai package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

[top].

- mvn life cycle commands.
- To run jar file generated after mvn package command and jar file generated in the target folder .this is also a way to run springboot project.
- Mvn install jar of current project is copied to m2 repo in local system .
- mvn clean all generated will get cleaned .
- if system donot have maven then also it can be run externally check commands if required .
- mvn package creates jar files after compilation.

-VAR files cannot be run like that we need to deply it on server.

Structure of SpringBoot Application

Main : core functiality

- java folder code
- resource to add like photos we used in it
- application properties to set Mongo Db configuratons .

pom.xml :

- dependencies of project
- plugins in projects help to package code in jar
- parent : koi bhi parent dedo use inherit krlega diniya bhar k plugins

Maven complier plugin ko bol rhe source code is in 22 version and after compilation it must be compatible with 22 version.

Jar of spring boot is different from comman jar – as have dependencies required to run this springboot project as well as compilled code .

- Normal jar just have compilled code .
- 2 jar are created once ORIGINAL is generated then it is repackaged to make new jar possible due to plugins.

INTERNAL WORKING OF SPRINGBOOT

1. Spring provides a functionality to create object of its own. We can say object creation can be done from externally i.e Inversion of control
2. Means hum khud create nahi kehte but spring boot se object lane kehte h called as INVERSION OF CONTROL.
3. Spring provides IOC container which have all classes of project then jo bhi classes ki object zarurat hogi woh de dega .
4. IOC/Application context container me bht sare objects pade hote h jab zarurat pade tab mang lete h.

HOW IOC Keps Object ?

- It scans the folder com.__.__
- If we have 100 classes it donot keps all but keps specific classes
- If @component is specified over class then only will keep class in the IOC container

Annotations are written over

- Class
- Interface
- Method
- Field

****Provides information jiske bhi upar likha hua h and depend ky information based upon kis par likha h.

@Component – Species class as Spring Bean .and came to IOC container.

Beans – If object came under the IOC container then object is called BEAN.

```
@SpringBootApplication
```

This annotation does 3 things of three annotations :

- @configuration
- @EnableAutoConfiguration -
- @ComponentScan – finds classes with @component inside base package com._._

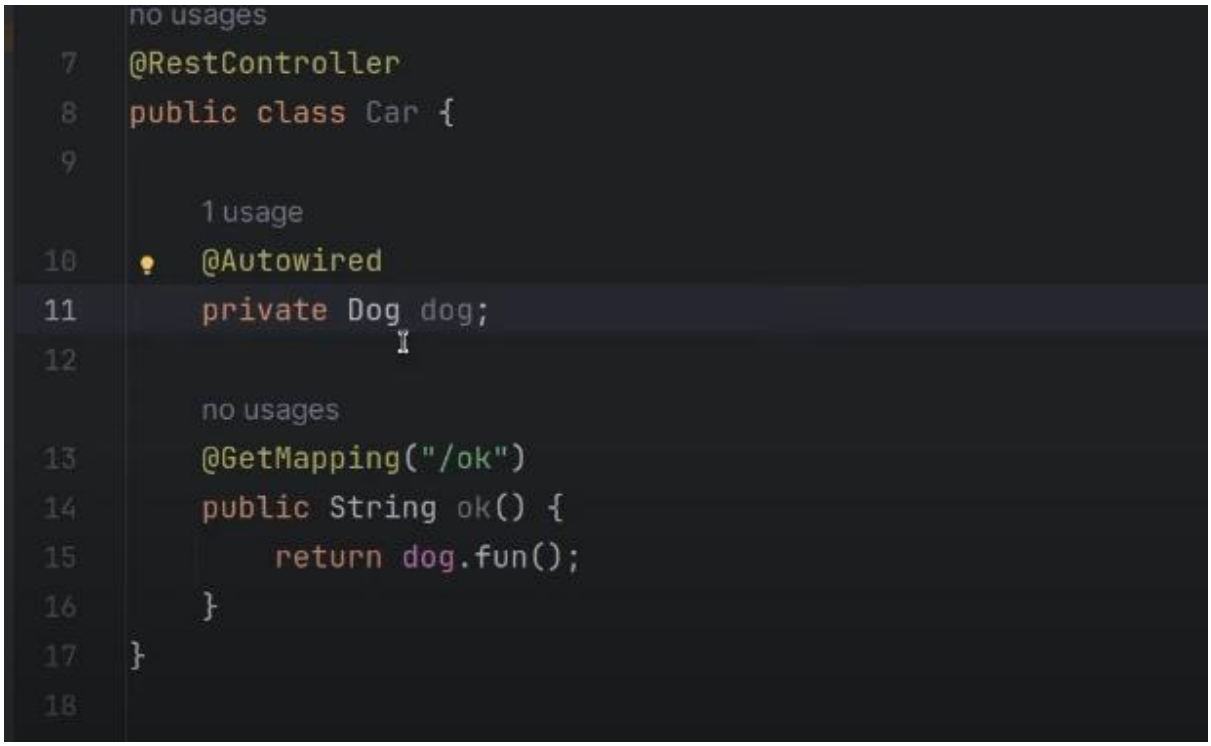
COMPONENT SCAN : @ComponentScan

@RestController – this perform functions of @Component + Something else .

@Autowired – dependency injection and IOC context keep it in bag and use it whenever required OR

We can say make Bean of object .

- Agar Hamare code me 100 classes – agar sab dog ka alag object banaege toh bht sare instances ban jaege toh @Autowired se ek object har jagah use hojaega .
- @Autowired help krta h jo present h bean class uske object ko specify krne ke liye.



ENABLE AUTO CONFIGURATION : @EnableAutoConfiguration

- Does configuration automatically.

- Ex : just DB server link user id password . Spring Boot does all it automatically .

CONNFIGURATION : @configuration

- Class provides some configuration
- Generally this annotation is used with some other annotations.

Errors 1 : mvn is not loading in local system due to which code is not running .

REST API – Representational State Transfer Application programming interface

REQUEST :

1. GET
2. POST
3. PUT
4. DELETE

Request involves : HTTP Verb(get,post,patch,delete) + URL

Annotations :

@RestController : this perform functions of @Component(means added to bean) + over class makes it api .

- Written over controller class

  @RestController
  Class Controller{

  }

- Every thing Returned in Json if specified this annotation over class

Explain MVC Architecture

Controller : will have the entry point and only the end point defined.

Services : will have logic written .

Controller : These are Special types of Component (Bean) that handles http request .

- Every time we develop backend via SpringBoot we should develop a health check controller .

For Get method Api :

- @GetMapping : used to map end point of get method to its method .

Similary ,

- @postMapping

***Every Method in controller should have to be public so that accessed by the http requests .

@PostMapping :

- Used to create end point of post request
- **@RequestBody (type of req_obj) :** take data from request and convert it in Java Objects .

Types of params in Spring :

- **@RequestParam** : to extract query parameters, form parameters, and even files from the request.
- **@PathVariable :** can be used to handle template variables in the request URI mapping, and set them as method parameters.

****What is main difference between @RequestParam and @PathVariable in Spring MVC controller?

If some verb has same URL application will not start ***if same URL rakhna h toh verb change krdo(request type change krdo)

***Difference between put and Patch ?

CRUD OPERATIONS :

```java
@RestController  no usages
@RequestMapping("/Blog")
public class BlogEntryController {

//      currently we donot have db so we created it as table //
        private Map<Long, Post_structure> Blog_posts = new HashMap<>();  5 usages

//      returns all posts in form of array list
        @GetMapping  no usages
        public List<Post_structure>get_all(){
            return new ArrayList<>(Blog_posts.values());
        }
//
        @PostMapping  no usages
        public String create_blog(@RequestBody Post_structure my_entry){
                Blog_posts.put(my_entry.getId(), my_entry);
                return "post created";
        }
```

```java
        @GetMapping("/id/{myID}")  no usages
        public Post_structure get_post_by_id(@PathVariable Long myID){
                return Blog_posts.get(myID);
        }

        //here if GetMapping hi rakha toh chalega hi nahi verb change krke DeleteMapping krna pdega
        @DeleteMapping("/id/{myID}")  no usages
        public String delete_post_by_id(@PathVariable Long myID){
                Blog_posts.remove(myID);
                return "Post Deleted Successfully";
        }

        @PutMapping("/id/{myID}")  no usages
        public String update_post_by_id(@PathVariable Long myID,@RequestBody Post_structure my_entry){
                Blog_posts.put(myID,my_entry);
                return "Post Updated Successfully";
        }
}
```

JDBC : Java DataBase Connectivity

YT : https://youtu.be/TcJZQvDE1ow?si=cRuAdT4K2m2cC-gj

- 1 Driver Class and 3 Interfaces
  - DeviceManager Class
  - Connection Interface
  - Statement And Prepared Statement Interface
  - Result Set Interface
- Program Flow:
  - Connect IDE with Database with necessary Connector:
    - MYSQL connector J to import in the Intelli J external libraies
  - Load Drivers
  - Create Connections
  - Create Statement
  - Execute Query
- There is only possibility of two Exceptions :
  - ClassNotFound
  - SqlException

Code : https://github.com/kaushaldeokar/JDBC-Boiler-and-CRUD-Ops/blob/main/src/Main.java

- Whenever retrive data : executeQuery(“”);
- Insert , update , delete

PREPARED STATEMENTS : query should be compiled only once but arguments should change. Before we have to write full query .

Code : https://github.com/kaushaldeokar/JDBC-Boiler-and-CRUD-Ops/blob/main/src/Connection_PreparedStatements.java

```java
//create Connections via Connection Interface
Connection connection = DriverManager.getConnection(Url, username, password);
//use Statement Interface and execute method of Connection interface
Statement statement = connection.createStatement();
// now we have to perform Crud Operation using this statement interface only
// **PREPARED STATEMENTS ** //
String Query = "INSERT INTO Students(name , age , marks) VALUES(?,?,?)";
PreparedStatement preparedQuery = connection.prepareStatement(Query);
preparedQuery.setString( parameterIndex: 1, x: "KD");
preparedQuery.setInt( parameterIndex: 2, x: 25);
preparedQuery.setDouble( parameterIndex: 3, x: 99.5);

int rows_added = preparedQuery.executeUpdate();

if(rows_added>0){
    System.out.println("Added Successfully");
}else System.out.println("FAILED");
```

ResultSet : Object used to store the data that is retrieved .

```java
//Retrival//
String Q2 = "select * from Students ";
PreparedStatement pre_stat = connection.prepareStatement(Q2);
ResultSet res = pre_stat.executeQuery();

while(res.next()){
    int id = res.getInt( columnLabel: "id");
    String name = res.getString( columnLabel: "name");
    System.out.println("ID : "+id);
    System.out.println("Name : "+name);
}
```

BATCH PROCESSING :

```java
while(true){
    String name = scan.nextLine();
    int age = scan.nextInt();
    double marks = scan.nextDouble();
    System.out.println("Want to enter more data (0/1) : ");
    int choice =scan.nextInt();

    String query = String.format("INSERT into Students (name , age , marks) Values ('%s',%d,%f)"
    statement.addBatch(query);

    if(choice==0)break;

}
//***will result in return of array ***//
// will result in boolean array particular row is added or not //
int[]  rows_added = statement.executeBatch();

boolean check_if_all_rows_added = true;
for ( int i=0 ;i<rows_added.length;i++){
    if(rows_added[i]==0){
        check_if_all_rows_added = false;
        System.out.println(i+1 + " data not added");
    }
}

if(check_if_all_rows_added == true){
    System.out.println(" data added Succesfully");
}
```

Code : https://github.com/kaushaldeokar/JDBC-Boiler-and-CRUD-Ops/blob/main/src/MySQL_Batch_Processing.java

TRANSACTION HANDLING :

1. Commit
2. Rollback

- **Used to ensure data consistency while the failed transaction .
- BEFORE when connection is setup the transaction is autoCommited.
- We only commit the connection and also rollback the Connection.
- If we find tranaction incomplete we rollback.

Connection.AutoCommit is by default true in JAVA

DB UNDERSTANDINGS

ORM : Object Relational Model

- It is a method to map java object to database tables
- Allows to interact with database using object oriented programming concepts help to interact with data base much easier .
- **Hum operation krege class par and operation database pr hojaega**
- Consider a Java class User and DataBase Table users .
- **ORM Framework like Hibernate can map fields to coloums of users table.**
- **Making easier to insert , update , retrieve , delete records.**

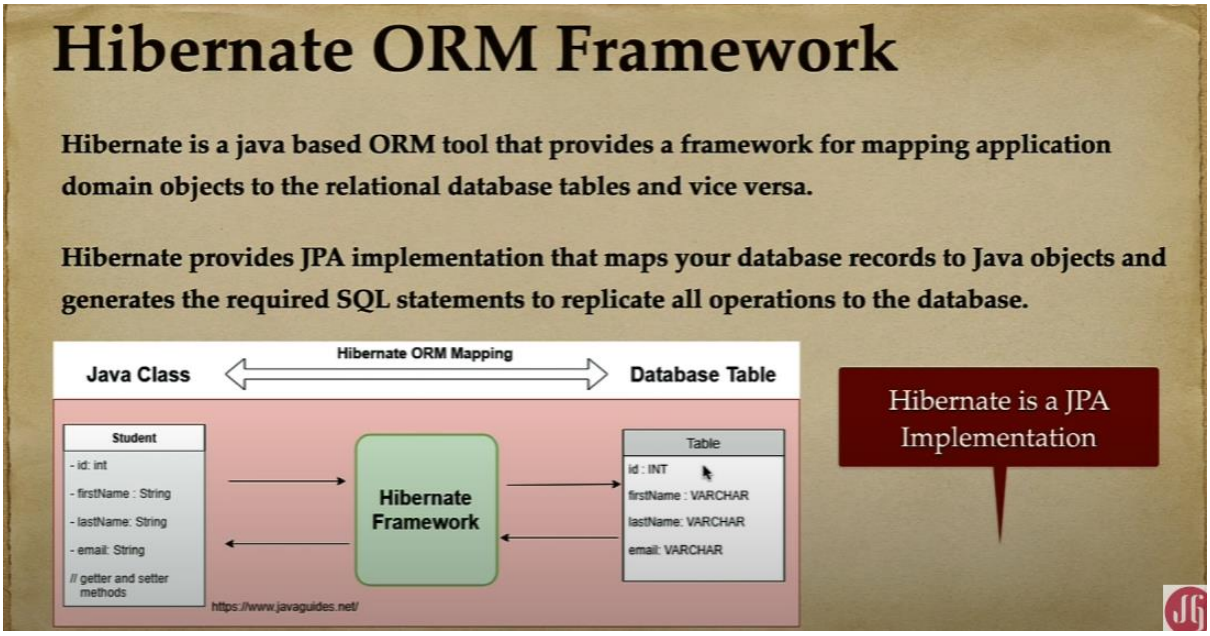
JPA : JAVA Persistance API

- **Set of Rules to achieve ORM .**
- Includes Interfaces and anotations tha we use in Java Classes , require a persistence provider (ORM Tools) for implementaation.

  ORM Tools / Persistance Provider :

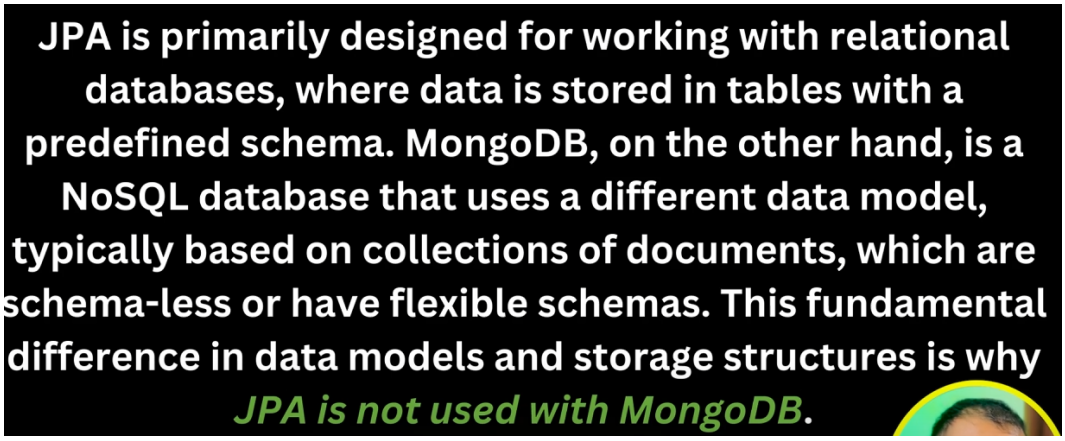
  To use JPA , you need a persistance provider . A persistence provider is a specific implementaion of JPA specification .
  Example of JPA Persistance provider are :
  - Hibernate
  - EclipseLink
  - OpenJPA
- The providers implement JPA interfaces and provide functionality to interact with databases.

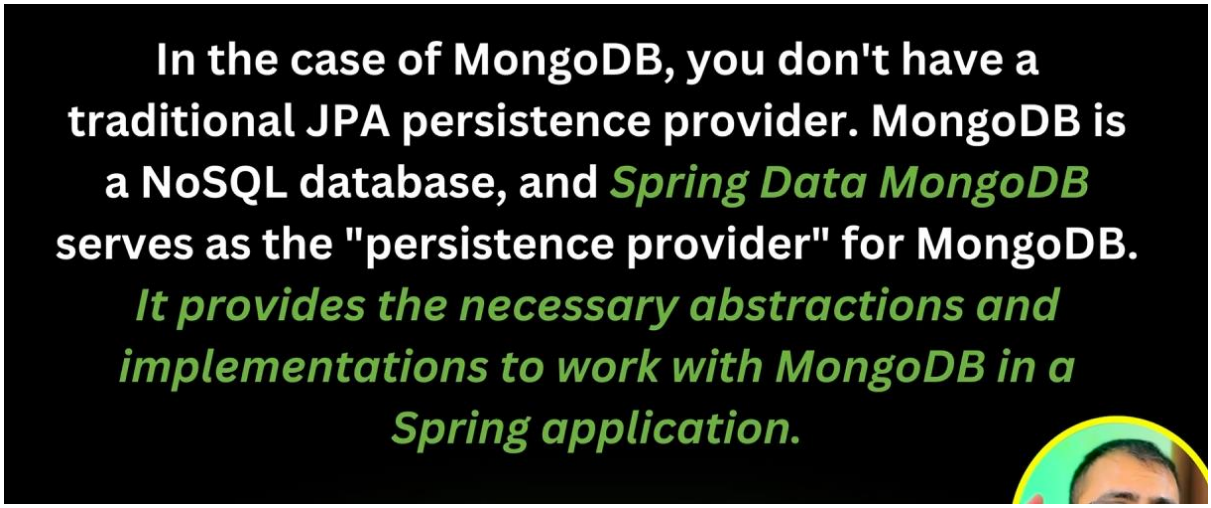
******IN ACTUAL WE WILL USE HIBERNATE**



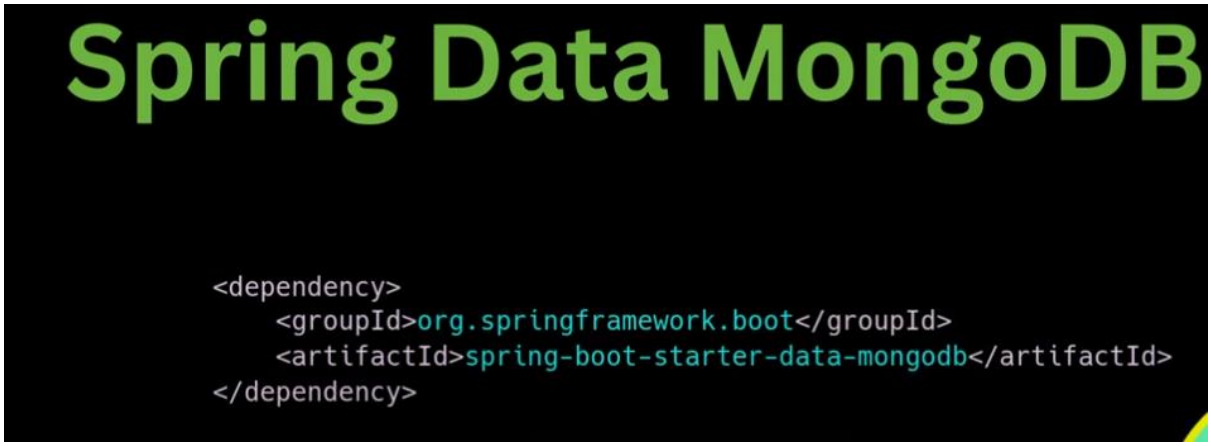
SPRING DATA JPA : built on top of jpa and hides complexity .

JPA WITH DATA BASES :





IN CASE OF MONGO_DB:



METHODS to INTERACT WITH DATABASES:

While using Spring data JPA for relational DataBases (MYSQL) and Spring Data MongoDb

Ther are two methods :

- **Query Method DSL:**
  More simple way to create queries based upon convention
- **Criteria API :**
  Provides dynamic and Programative approach to complex and custon queries .

**Spring Data JPA is part of spring framework that simplifies data access

**Spring Data MongoDb does same in case of MongoDb

Spring Connect to MySql data Base :

****Configurations**

```
spring.jpa.hibernate.ddl-auto = update
#alters table according to the jpa enitity
spring.datasource.url=jdbc:mysql://localhost:3306/blog
spring.datasource.username=root
spring.datasource.password=Kd123@123
#This "spring.jpa.database-platform" specify which database
# you are going to use. It may be postgress, MySQL etc. According to that you need to specify it.
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver


spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
#will show all statement hibernate create behind the scene and must be formatted
```

```
<dependencies>

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


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

```
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-test</artifactId>
       <scope>test</scope>
    </dependency>

    <dependency>
       <groupId>com.mysql</groupId>
       <artifactId>mysql-connector-j</artifactId>
       <scope>runtime</scope>
    </dependency>

</dependencies>
```

Spring JPA Annotations : https://www.digitalocean.com/community/tutorials/jpa-hibernate-annotations

1. `javax.persistence.Entity`: Specifies that the class is an entity. This annotation can be applied on Class, Interface of Enums.

```
import javax.persistence.Entity;

@Entity
public class Employee implements Serializable {
}
```

2. `@Table`: It specifies the table in the database with which this entity is mapped. In the example below the data will be stores in the “employee” table. Name attribute of @Table annotation is used to specify the table name.

```
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "employee")
public class Employee implements Serializable {
}
```

4. `@Id`: This annotation specifies the primary key of the entity.

```
import javax.persistence.*;

@Entity
@Table(name = "employee")
public class Employee implements Serializable {
  @Id
  @Column(name = "id")
  private int id;
}
```

5. `@GeneratedValue`: This annotation specifies the generation strategies for the values of primary keys.

```
import javax.persistence.*;

@Entity
@Table(name = "employee")
public class Employee implements Serializable {

  @Id
  @Column(name = "id")
  @GeneratedValue(strategy=SEQUENCE, generator="ID_SEQ")
  private int id;
}
```

```
@Table(
   name = "product_serial_group_mask",
   uniqueConstraints = {@UniqueConstraint(columnNames = {"mask", "group"})}
)
```

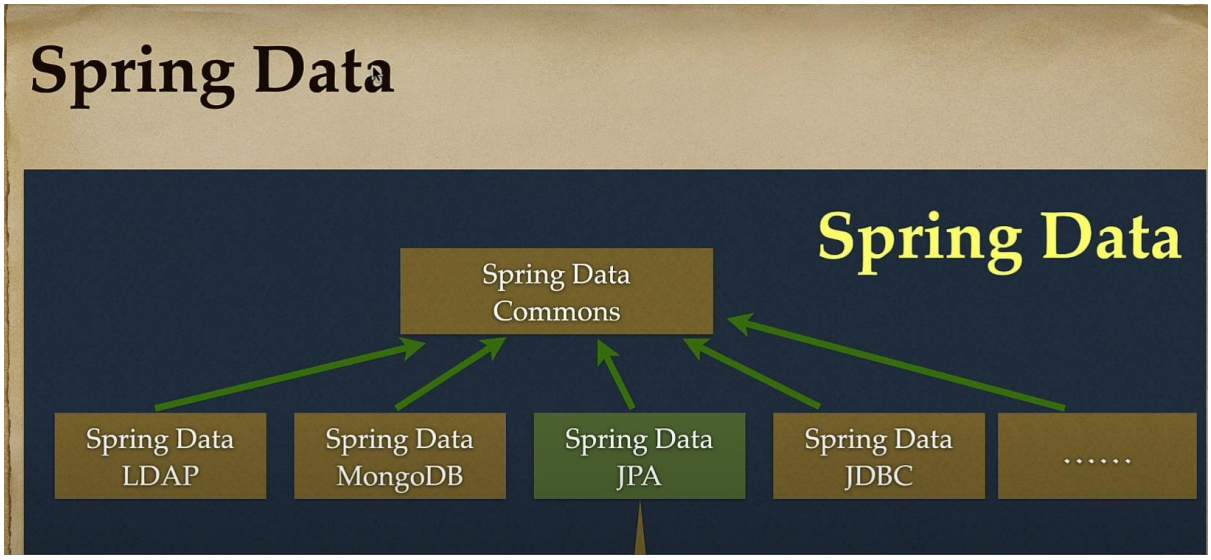
Implies that the values of mask + group combined should be unique. That means you can have, for example, a record with **mask.id = 1** and **group.id = 1**, and if you try to insert another record with **mask.id = 1** and **group.id = 2**, it'll be inserted successfully, whereas in the first case it wouldn't.

If you'd like to have both mask and group to be unique separately and to that at class level, you'd have to write the code as following:

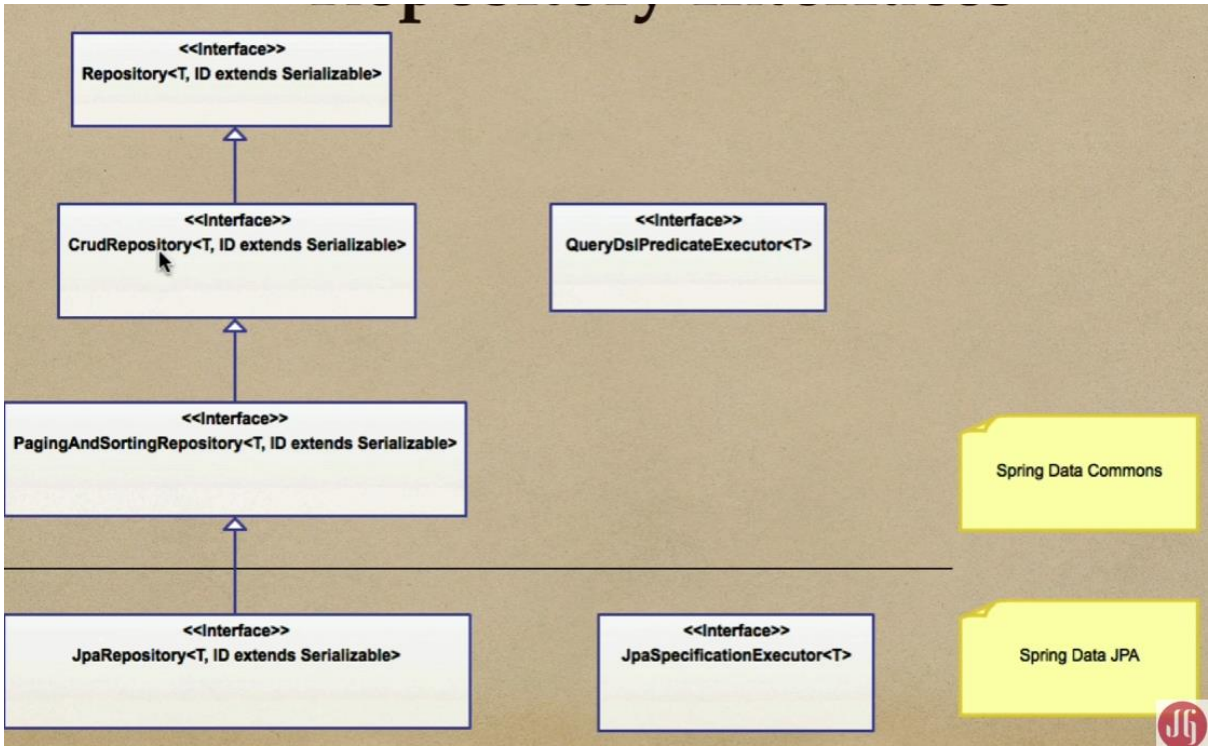
```
@Table(
        name = "product_serial_group_mask",
        uniqueConstraints = {
                @UniqueConstraint(columnNames = "mask"),
                @UniqueConstraint(columnNames = "group")
        }
)
```

This has the same effect as the first code block.

@GenratedValue : 4 method to do this . IDENTITY set auto increment



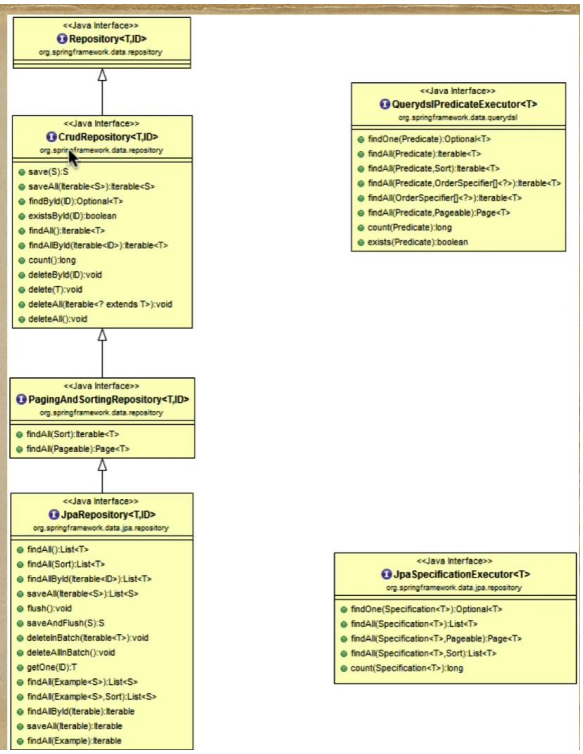
Respository Interface



Repository

```
ProductRepository extends JpaRepository<Product,Integer> {
```

Understanding JpaRepository Interface



JpaRepository Implementation

SimpleJpaRepository implementation class provides implementation for methods

```
@Repository
@Transactional(
    readOnly = true
)
public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {
    private static final String ID_MUST_NOT_BE_NULL = "The given id must not be null!";
    private final JpaEntityInformation<T, ?> entityInformation;
    private final EntityManager em;
    private final PersistenceProvider provider;
    @Nullable
    private CrudMethodMetadata metadata;
    private EscapeCharacter escapeCharacter;
```

JpaRepository Interface

```
@NoRepositoryBean
public interface JpaRepositoryImplementation<T, ID> extends JpaRepository<T, ID>, JpaSpecificationExecutor<T> {
    void setRepositoryMethodMetadata(CrudMethodMetadata crudMethodMetadata);

    default void setEscapeCharacter(EscapeCharacter escapeCharacter) {
    }
}
```

Steps to create and use Spring Data JPA Repository

1. Create a repository interface and extend to JpaRepository interface

2. Add custom query methods to the created repository interface (if we need them)

3. Inject the repository interface to another component and use the implementation that is provided automatically by Spring Data Jpa.

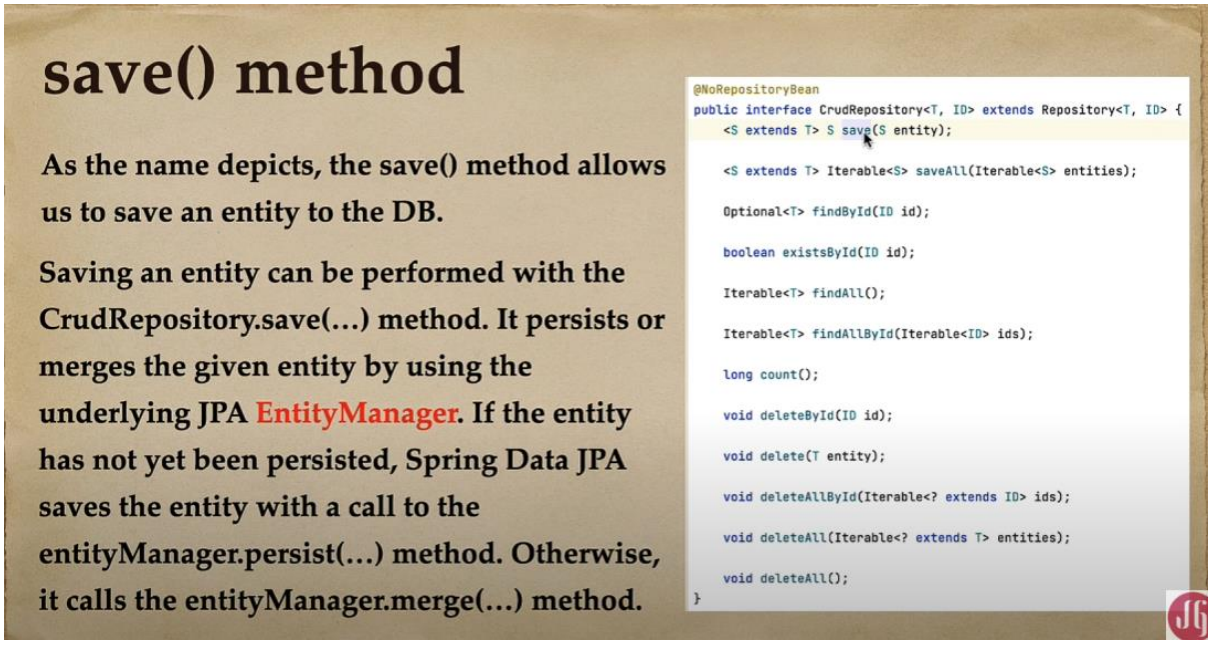
1. Create a repository interface and extend to JpaRepository interface

```
public interface ProductRepository extends JpaRepository<Product,Integer> {

}
```

JPA Entity

Primary Key

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "PRODUCT_TBL")
public class Product {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private int quantity;
    private double price;
}
```



Annotation used in Repository :

@Repository :

- Version of @component + (Something)
- Used to specify data layer compnent

Anotation used in Service Package :

@Service :

- Version of @component + (Something)
- Used to specify data layer compnent

@Transactional

- **Atomicity**: Ensures that all operations within a transaction either complete successfully or none of them do. This is critical for maintaining data integrity.
- **Consistency**: Guarantees that the database remains in a consistent state before and after the transaction.
- **Isolation**: Controls the visibility of changes made by one transaction to other transactions. It ensures that the intermediate state of a transaction is invisible to other transactions until it is complete.
- **Durability**: Ensures that once a transaction is committed, its changes are permanent and survive system crashes.