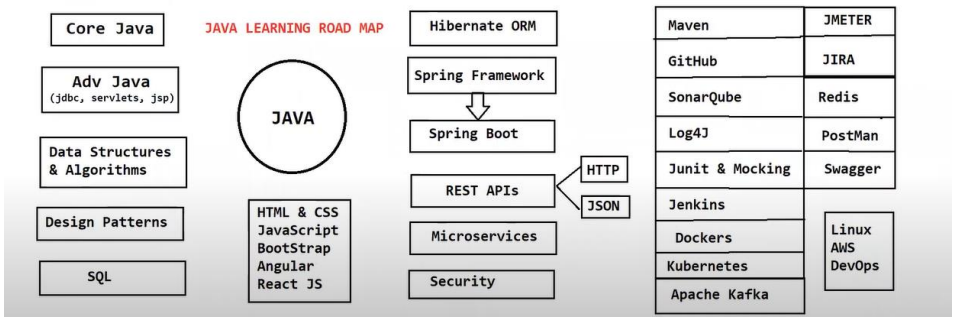
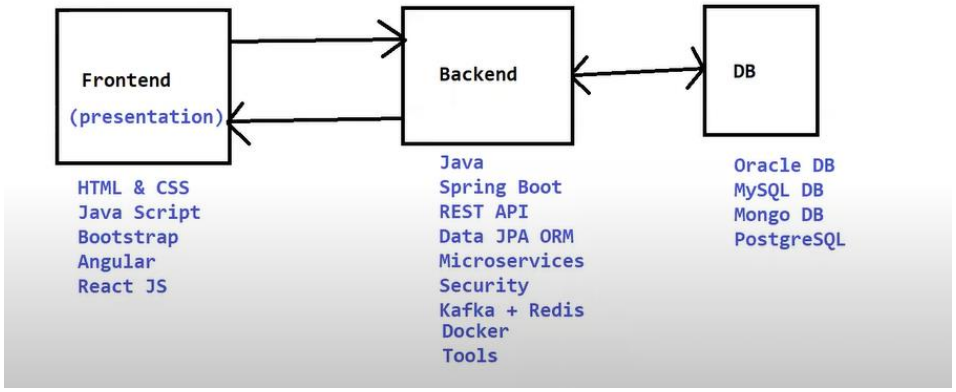




Application Development

Application development divided into 2 parts

i) Frontend Development
ii) Backend Development

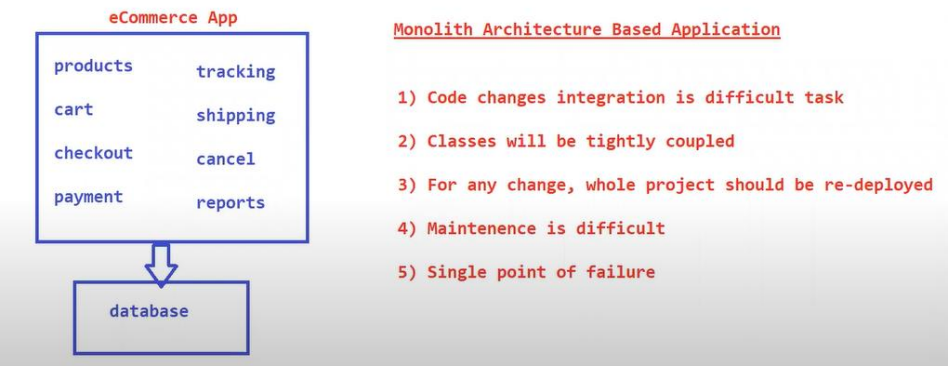
Full-stack Development = Frontend development + Backend development

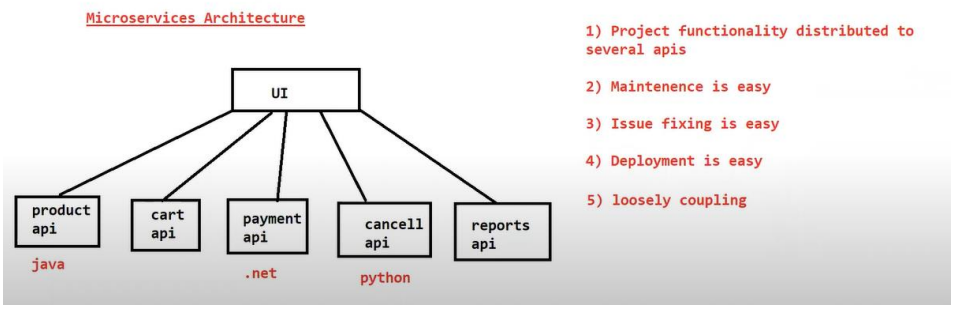
Backend Development: It contains business logic of our application.

(Webserver calls, Validations, email logic, DB communication)

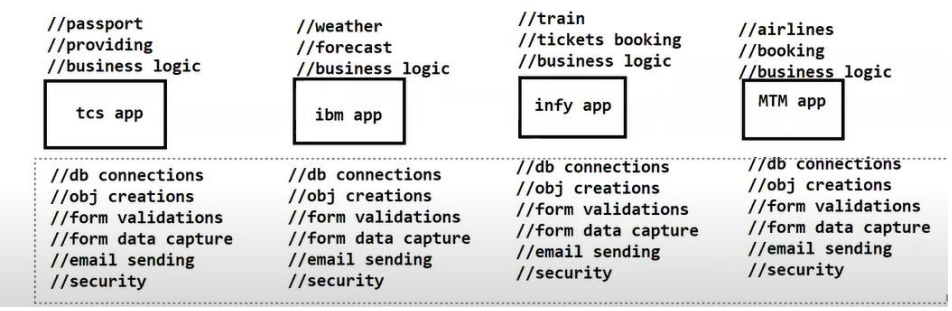
Frontend Development: It contains user interface of the applications.

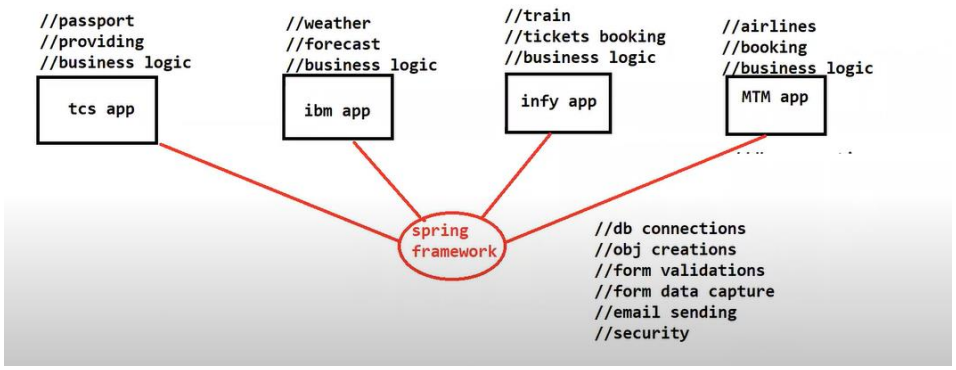
(Presentation logic)





Without Framework we need to write all the **common logic** again and again from the beginning.





With Framework we don't need write common logic, it reduces our effort and increase time **so that we can only focus on our business logic**.

- Framework means semi-developed software.
- Frameworks provides some common logics which are required for application development

  Ex: Hibernate, Struts, Spring etc...

- If we use framework in our application development then we need focus only on business logic (framework will provide common logics)

- **Hibernate**: It is an ORM framework. Used to develop persistence layer of our application

- **Struts**: It is a web framework. Used to develop web layer of our application.

- **Spring**: It is an application development framework. Entire project can be developed by using this. (We can do application end to end development)

Advantage of Spring Framework

- It is a free & open-source framework
- Spring is very light weight framework
- Spring is versatile framework

(Spring can be integrated with any other java framework available in the market)

- Spring is non-invasive framework

(Spring framework will not force us to use framework related interfaces or classes)

Ex: To create a servlet we need to implement **Servlet Interface** or we need to extend **HttpServlet** or **GenericServlet**. That means servlet is forcing us to use Servlets specific **interface** or **classes**.

Note: In Spring we can create a simple **POJO** and we can ask spring to execute our **POJO**

- Spring works based on POJO and POJI model

Spring Modules

1) Spring Core

2) Spring Context

3) Spring DAO / Spring JDBC

5) Spring AOP

6) Spring ORM

7) Spring Web MVC

8) Spring Security

9) Spring REST

10) Spring Data

11) Spring Cloud

12) Spring Batch etc...

Note: Spring framework is loosely coupled. It will not force to use all modules.

Based on project requirement we can choose which modules we need to use from Spring.

Note: **Spring Core** is the base module for all other modules in Spring. To work with any module in spring first we need to know Spring Core module.

- **Spring Core** Module is base module in Spring Framework. It is providing IOC container & Dependency Injection.

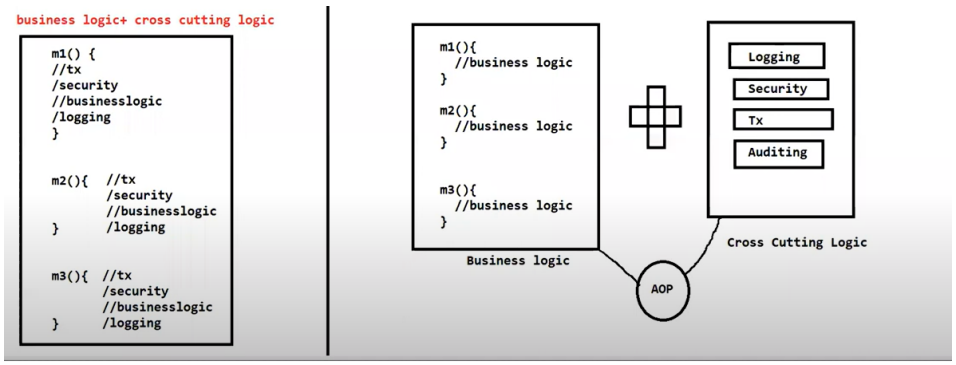
Note: **IOC** & **DI** are fundamental concepts of **Spring Framework**.

- **Spring Context** module will deal the **configuration related stuff**

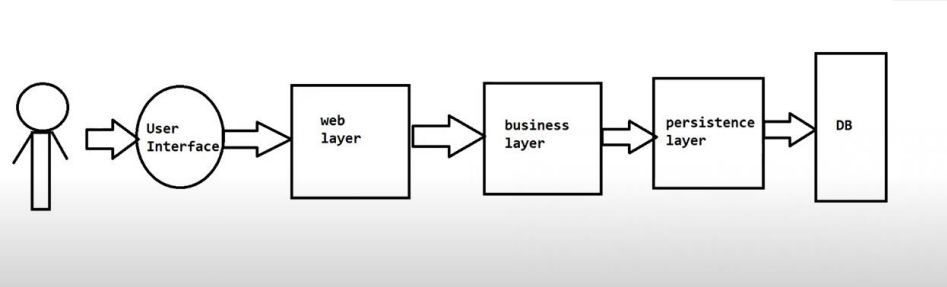
- **AOP** stands for **Aspect Oriented Programming**. Spring AOP is used deal with **Cross Cutting logics** in application.

Application = Business Logic + Cross Cutting Logic

Note: We can separate business logic and cross cutting logic using AOP module.



- **Spring JDBC** / Spring DAO module used to develop Persistence Layer
- **Spring ORM** module is used to develop Persistence Layer with ORM features.
- **Spring Web MVC** Module is used to develop Web Applications.
- **Spring Security** module is used to implement Security Features in our application (Authentication & Authorization)
- **Spring REST** is used to develop RESTful services (REST API)
- **Spring Data** is used to develop persistence layer. It provided pre-defined repositories to simplify CRUD operations.
- **Spring Cloud** providing Cloud concepts like Service Registry, Cloud Gateway, Filters, Routing, FeignClient etc..
- **Spring Batch** is used to implement Batch Processing in our application. Batch Processing means bulk operation.



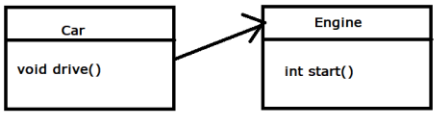
Sample Project Structure may look like this

We can build whole project by using Spring Framework. It provides UI, Web Controller and Persistence Module also. With this one framework we can build end-end application.

Spring Core

- Spring Core is **base module in Spring** Framework
- All the other modules of Spring are developed on top of Spring Core Module only
- Spring Core providing fundamental concepts of Spring Framework (**IOC Container** & **Dependency Injection**)
- Spring **Core is all about Managing dependencies among the classes** (**Creating Objects** & **Injecting Objects**)

Dependency Injection

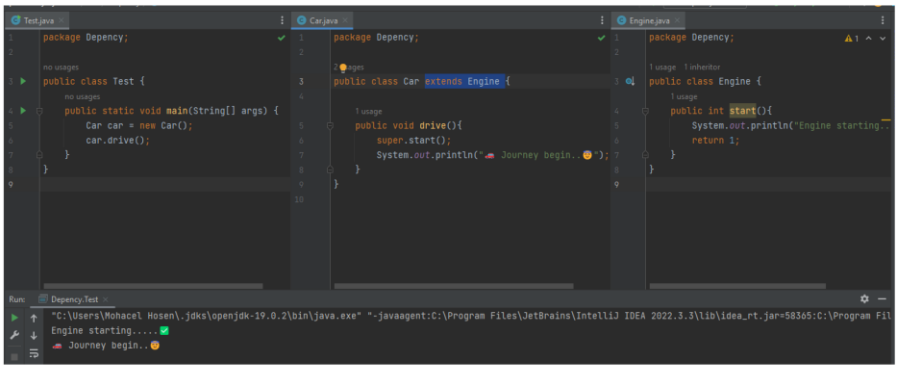


There are 2 ways that Car class can call Engine class method

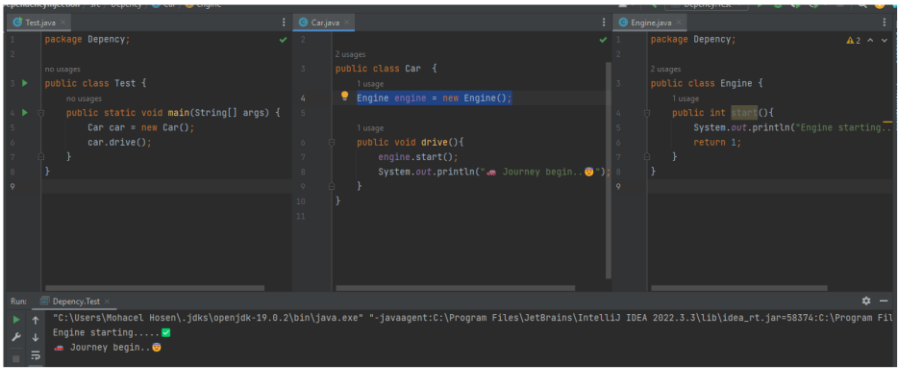
1) Inheritance
2) Composition (by creating object)

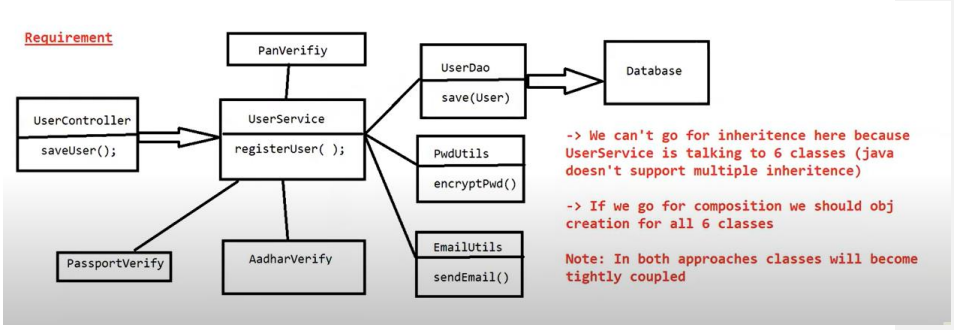
Both approch are not recommended they are tightly coupled. Besides Java does't support multiple inheritance

Inheritance Approch



Composition Approch





- In our application several classes will be available
- One class method wants to talk another class method
- We can establish communication among the classes in 2 ways
    1. Inheritance
    2. Composition
- If we use Inheritance or Composition then our classes will become tightly coupled.
- Instead of we are creating objects we can ask Spring Core to manage dependencies among our classes
- If we want Spring Core Module to manage dependencies among the classes with loosely coupling then we have to develop our classes by following some best practices.

Spring Core module suggesting Developer to follow "**Strategy Design Pattern**" to develop classes so that **Spring Core** can easily manage dependencies among the classes with loosely coupling.

Strategy Design Pattern

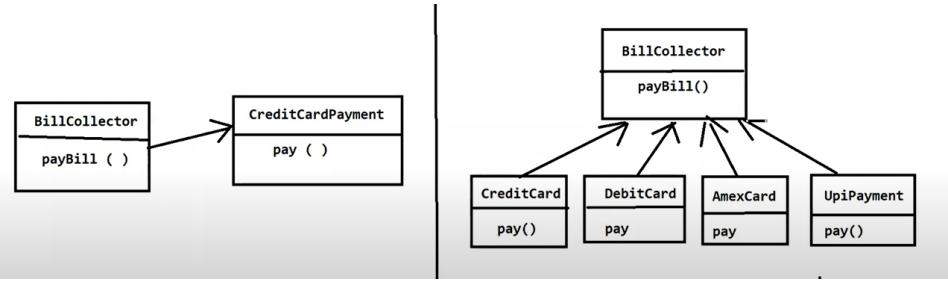
- It comes under Behavioral Design Pattern
- It enables our code to choose an algorithm at run time
- When we have multiple algorithms then we can use this pattern to choose one algorithm at run time

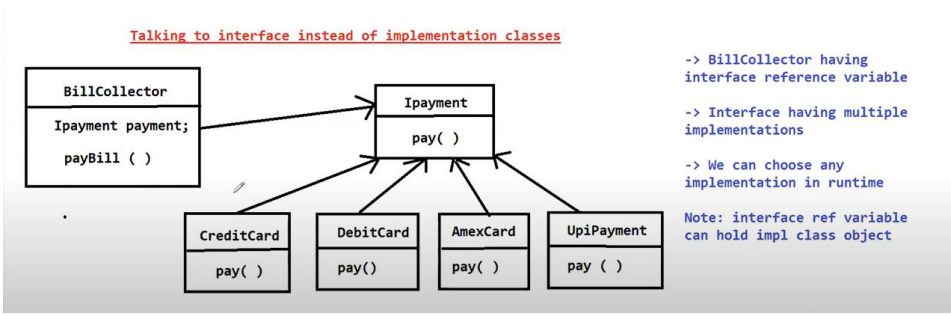
Rules

1) Favor Composition over inheritance

2) Code to interfaces instead of implementation classes

3) Code should be open for extension and code should be closed for modification





```java
package Payment;

public interface Ipayment {
    void pay (double amount);
}
```

Ipayment.java

```java
package Payment;

public class BkashPay implements Ipayment {
    @Override
    public void pay (double amount) {
        System.out.println("You paid "+ amount+ " successfully through Bkash Pay");
   }
}
```

BkashPay.java

```java
package Payment;

public class CreditCard implements Ipayment {

    @Override
    public void pay (double amount) {
        System.out.println("You paid "+ amount+ " successfully through Credit
card");
   }
}
```

CreditCard.java

```java
package Payment;

public class DebitCard implements Ipayment{
    @Override
    public void pay(double amount) {
        System.out.println("You paid "+ amount+ " successfully through Debit
card");
    }
}
```

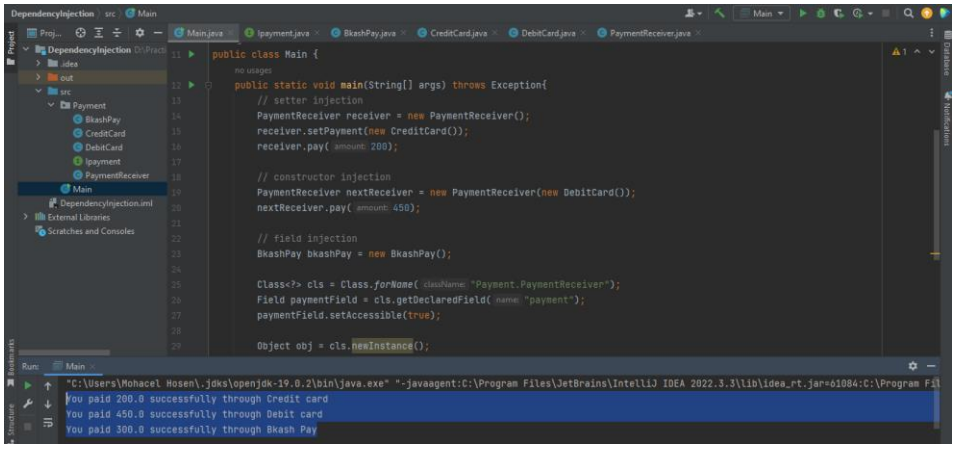
```java
package Payment;

public class PaymentReceiver {
    private Ipayment payment;
    // no args constructor
    public PaymentReceiver(){super();}
    // all args constructor
    public PaymentReceiver(Ipayment payment){this.payment=payment;}

    // setter method for instance variable(payment)
    public void setPayment(Ipayment payment){
        this.payment = payment;
    }
    //normal method
    public void pay(double amount){
        payment.pay(amount);
    }
}
```

PaymentReceiver.java

```java
import Payment.BkashPay;
import Payment.CreditCard;
import Payment.DebitCard;
import Payment.PaymentReceiver;

import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws Exception{
        // setter injection
        PaymentReceiver receiver = new PaymentReceiver();
        receiver.setPayment(new CreditCard());
        receiver.pay(200);

        // constructor injection
        PaymentReceiver nextReceiver = new PaymentReceiver(new DebitCard());
        nextReceiver.pay(450);

        // field injection
        BkashPay bkashPay = new BkashPay();

        Class<?> cls = Class.forName("Payment.PaymentReceiver");
        Field paymentField = cls.getDeclaredField("payment");
        paymentField.setAccessible(true);

        Object obj = cls.newInstance();
```

```
        PaymentReceiver paymentReceiver = (PaymentReceiver) obj;
        paymentField.set(obj, bkashPay);
        paymentReceiver.pay(300);


    }
}
```

Main.java



Dependency Injection

- The process of injecting one class object into another class object is called as Dependency Injection

- We can perform Dependency Injection in 3 ways

  1) Setter Injection
  2) Constructor Injection
  3) Field Injection

- The process of injecting one class object into another class object using Setter method then it is called as Setter Injection.

Ex::

```
// setter injection
PaymentReceiver receiver = new PaymentReceiver ();
receiver.setPayment (new CreditCard ());
receiver.pay (200);
```

- The process of injecting one class object into another class object using Constructor is called as Constructor Injection

Ex::

```
// constructor injection
PaymentReceiver nextReceiver = new PaymentReceiver (new DebitCard ());
nextReceiver.pay (450);
```

- The process of injecting one class object into another class object using variable is called as Field Injection.

```
// field injection
BkashPay bkashPay = new BkashPay();

Class<?> cls = Class.forName("Payment.PaymentReceiver");
Field paymentField = cls.getDeclaredField("payment");
paymentField.setAccessible(true);

Object obj = cls.newInstance();
PaymentReceiver paymentReceiver = (PaymentReceiver) obj;
paymentField.set(obj, bkashPay);
paymentReceiver.pay(300);
```

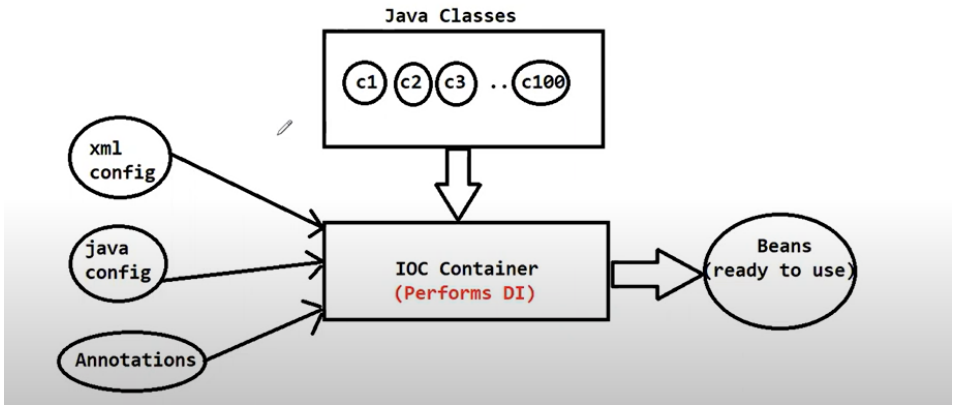
Note: If variable is declared as public then we can access that variable outside of the class we can initialize directly.

Note: If the variable declared as private then we can't access that variable outside of the class directly. To access private variable outside of the class we can use Reflection API.

- If we develop **the project without using spring fra**mework then **programmer should perform dependency injection** among the classes in the application.
- **If programmer performs dependency** injection, then classes will **become tightly coupled**.
- In Realtime project we will have 100's of classes then performing Dependency Injection is very difficult and code will become cumbersome.
- To overcome this problem, we can use Spring IOC container.
- If we develop the project using spring framework then **Spring IOC** will **take care of Dependency Injections** in our application.
- **IOC is a principle** which is responsible to **manage and collaborate dependencies** among the classes available in the application.

Note: IOC stands for Inversion of Control. DI stands for Dependency Injection.

In Spring Framework IOC will perform DI.



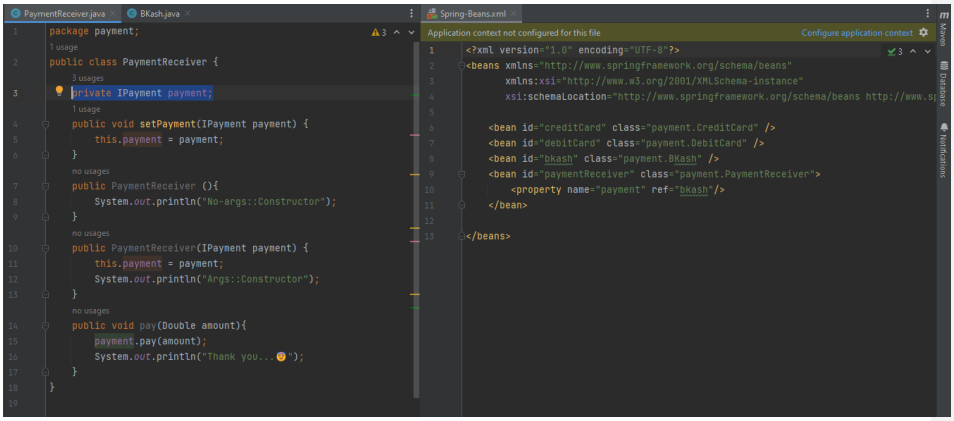
We need provide some information about our target class and dependent class so that IOC can perform DI.

Spring Context

1. Create a maven project without any architype
2. Add Spring context into pom file
3. Create **PaymentReceiver** Project using **Strategy Design pattern** to perform **dependency injection**.
   i. IPayment.java
   ii. CreditCard.java
   iii. DebitCard.java
   iv. Bkash.java
   v. PaymentReciver.java
   vi. Test.java

4. Create **Spring-Beans.xml** file for spring configuration ()
5. Create bean for target class and dependent class. Dependent class bean use property tag (**property is setter () injection**). So that our classes can be treated as Spring Beans

We are creating bean for **“private Ipayment payment”**




```
<bean id="creditCard" class="payment.CreditCard" />
<bean id="debitCard" class="payment.DebitCard" />
<bean id="bkash" class="payment.BKash" />
<bean id="paymentReceiver" class="payment.PaymentReceiver">
    <property name="payment" ref="bkash"/>
</bean>
```

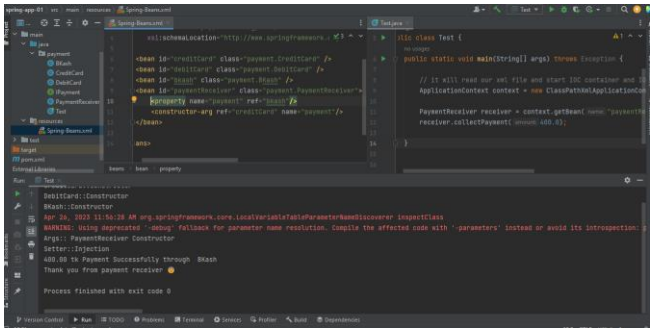
Setter Injection (property tag)

```
<bean id="creditCard" class="payment.CreditCard" />
<bean id="debitCard" class="payment.DebitCard" />
<bean id="bkash" class="payment.BKash" />
<bean id="paymentReceiver" class="payment.PaymentReceiver">
    <constructor-arg ref="creditCard" name="payment"/>
</bean>
```

Constructor Injection (constructor-args tag)

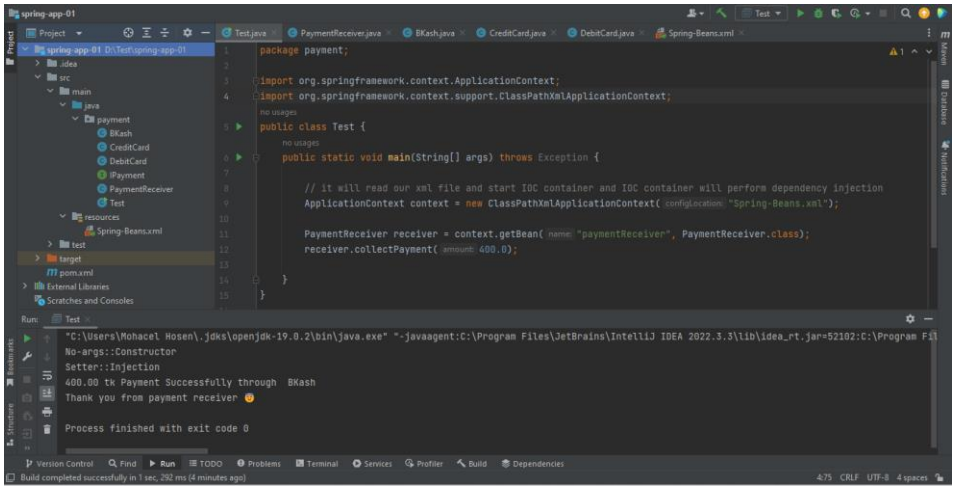
Note: When we perform both setter injection and constructor injection for the same variable then setter injection will override the constructor injection.

```
<bean id="paymentReceiver" class="payment.PaymentReceiver">
    <property name="payment" ref="bkash"/>
    <constructor-arg ref="creditCard" name="payment"/>
</bean>
```



6. Create **ApplicationContext(staring IOC container)** and get the bean object using **bean id**.

```
// it will read our xml file and start IOC container and IOC
container will perform dependency injection according to our
bean configuration
ApplicationContext context = new
ClassPathXmlApplicationContext("Spring-Beans.xml");

PaymentReceiver receiver = context.getBean("paymentReceiver",
PaymentReceiver.class);

receiver.collectPayment(400.0);
```

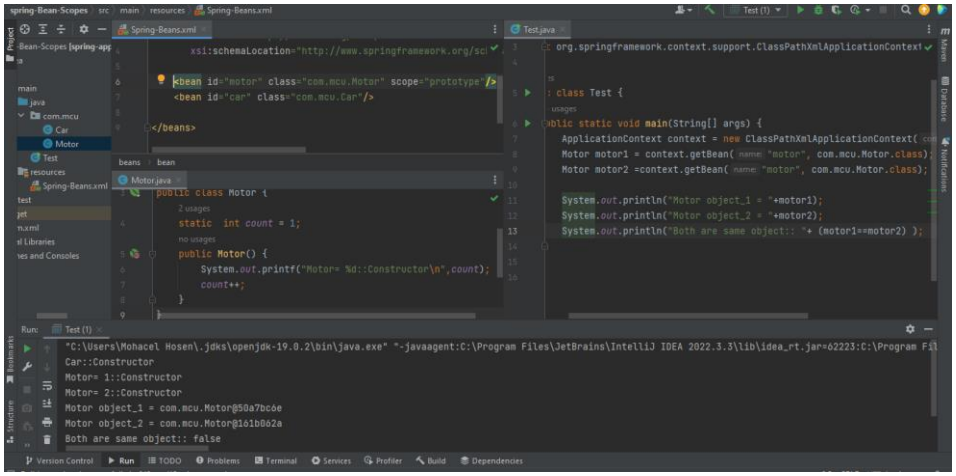


Bean Scopes

- Bean scope will decide how many objects should be created for a spring bean
- The default scope of spring bean is singleton (that means only one object will be created)
- We can configure below scopes for spring bean
  1) **singleton**
  2) **prototype**
  3) **request**
  4) **session**

Note: For **singleton beans** objects will be **created when IOC starts**

- When we call context.getBean(..) method then object will be created for prototype beans
- For prototype beans every time new object will be created



Autowiring

- In application several classes will be available
- One class wants to talk to another class
- We are using IOC container to perform that dependency injection
- We are giving instruction to IOC to inject dependent object into target object using 'ref' attribute

```
<bean id="paymentReceiver" class="payment.PaymentReceiver">
  <constructor-arg ref="creditCard" name="payment"/>
</bean>
```

- Using "ref" attribute we are telling to IOC which object it has to inject
- This process is called as Manual Wiring

```
package com.mcu;

public class Car {
    private DieselEngine engine;
    public Car() {
        System.out.println("Constructor:: Car");
    }

    public void setEngine(DieselEngine engine) {
        this.engine = engine;
    }
    public void drive(){
        if (engine.start()>=1){
            System.out.println("Journey start 😀....");
        }else{
            System.out.println("Engine start to fail  😞");
        }
    }
}
```

```
package com.mcu;

public class DieselEngine {
    public DieselEngine(){
        System.out.println("Constructor:: Diesel Engine");
    }

    public  int start(){
        //logic
        System.out.println("Diesel Engine Staring....");
        return 1;
    }
}
```

```
<bean id="engine" class="com.mcu.DieselEngine"/>

<bean id="car" class="com.mcu.Car" autowire="byName"/>
```

When Bean id match with varialbe name then "byName" auto-wiring will perform

ByName auto-wiring will happen if **variable_Name** and **bean id name** are match

```java
package com.mcu;

public class Car {
    private DieselEngine engine;
    public Car() {
        System.out.println("Constructor:: Car");
    }

    public void setEngine(DieselEngine engine) {
        this.engine = engine;
    }
    public void drive(){
        if (engine.start()>=1){
            System.out.println("Journey start 😀....");
        }else{
            System.out.println("Engine start to fail  😞");
        }
    }
}
```

Ref. Type

Ref. Name

```java
package com.mcu;

public class DieselEngine {
    public DieselEngine(){
        System.out.println("Constructor:: Diesel Engine");
    }

    public  int start(){
        //logic
        System.out.println("Diesel Engine Staring....");
        return 1;
    }
}
```

Ref. Name Ref. Type

```bean
<bean id="engine" class="com.mcu.DieselEngine"/>

<bean id="car" class="com.mcu.Car" autowire="byType"/>
```

When we have Reference Type bean then we can perform "byType" auto-wiring

byType auto-wiring will perform if **Reference_Type** and **Bean_Class** are match

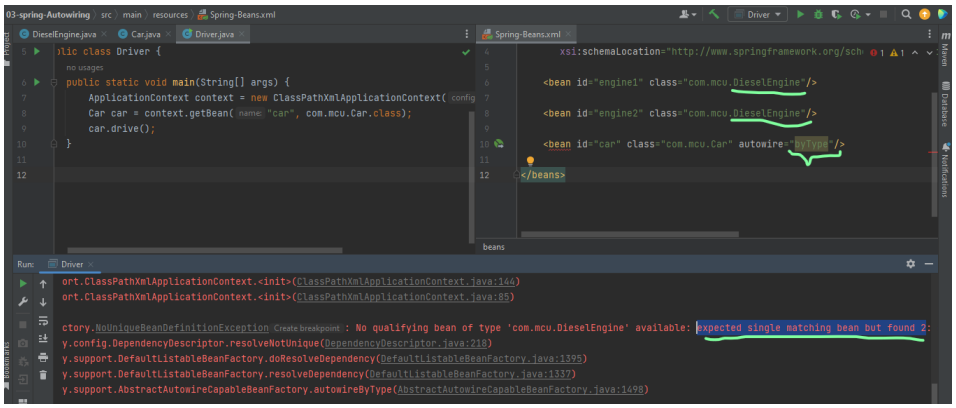
- Spring IOC supports Autowiring concept that means Spring IOC having capability to identify the dependent and inject dependent into target

- Autowiring having mode
    1) **by Name**
    2) **by Type**
    3) **constructor**
    4) **no**

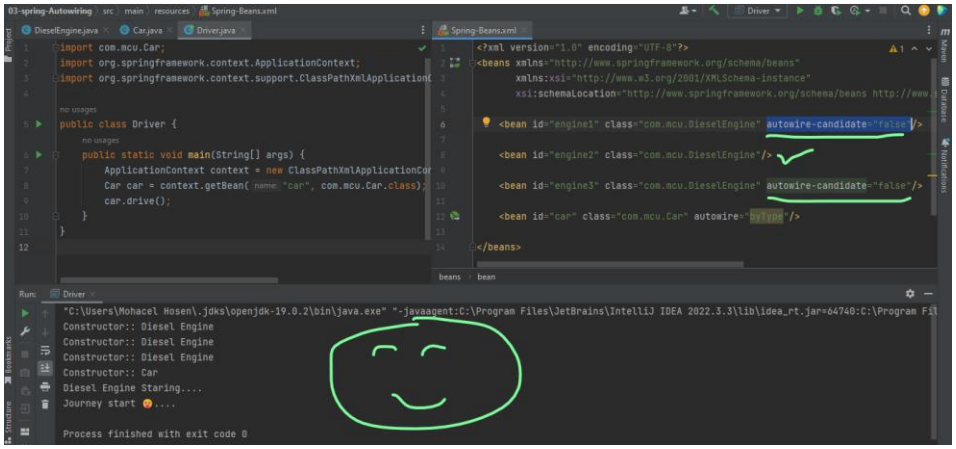
- **byName** means if target class variable name matched with any bean id/name in bean configuration file then IOC will consider that as dependent bean and it will inject that dependent bean object into target object.

- **byType** means it will check data type of the variable. With Datatype of variable if any bean class is configured then it will identify that as dependent and it will inject into target.

Note: We can configure one class for multiple times with different ids then we will get ambiguity problem in by Type scenario.

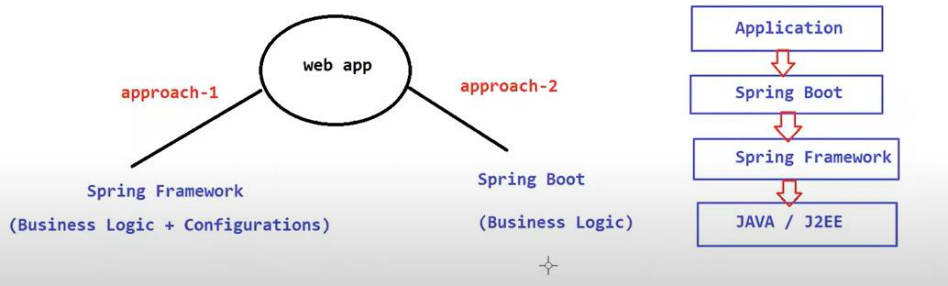


To overcome ambiguity problem, we need to use **autowire-candidate="false"**



Note: When we develop the application by using spring, programmer is responsible to take care of configurations required for the application.

Spring Boot



- It is another approach to develop spring-based applications with less configurations
- Spring Boot is an enhancement of Spring framework
- Spring Boot internally uses Spring framework only
- What type of applications we can develop using spring framework, same type of applications can be developed by using Spring boot also

Spring Boot = **Spring Framework - XML Configurations**

Spring Boot Advantages

1) **Auto Configuration**

2) **POM starters**

3) **Embedded Servers**

4) **Rapid Development**

5) **Actuators**

- Autoconfiguration means boot will identify the configuration required for our application and it will provide that configuration

    a) **Starting IOC container**
    b) **Creating Connection Pool**
    c) **Creation SMTP Connections**
    d) **Web Application Deployment**
    e) **Dependency Injections** etc....

- POM starters are nothing but dependencies that we use to develop our application

    a) **web-starter**
    b) **jdbc-starter**
    c) **security-starter**
    d) **mail-starter**

- Boot will provide web server to run our web applications. Those servers are called as Embedded Server

    a) **Tomcat**
    b) **Jetty**
    c) **Natty**

- Rapid Development means fast development. We can focus only on business logic

- Actuators are used to monitor and manage our application

    a) **How many classes loaded**
    b) **How many objects are created**
    c) **How many threads are running**
    d) **URL Mapping Available etc……**

Spring Boot Application Folder Structure:

```
Spring-Boot-App  ---------- Project Name (root folder)

  - src/main/java
            - Application.java   (This is start class)

  - src/main/resources
            - application.properties

  - src/test/java
            - ApplicationTests.java

  - Maven Dependencies   (It contains jars which got downloaded)

  - target  (it contains .class files)

  - pom.xml (Maven configuration file)
```

- **We will write our source code under src/main/java folder**
- **We will create our configuration files under src/main/resources folder**
- **We will create Junit class under src/test/java folder**
- **Project decencies will configure in pom.xml file**

  POM = Project Object Model

Note: When we **create boot application** by **default one java class will be created** with a **name Application.java** i.e., called **as Start class of spring boot**

❖ Start class is the entry point for boot application execution

```
package com.mcu.springboot;

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

@SpringBootApplication
public class Application {

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

}
```

➢ **@SpringBootApplication** annotation is equal to below 3 annotations

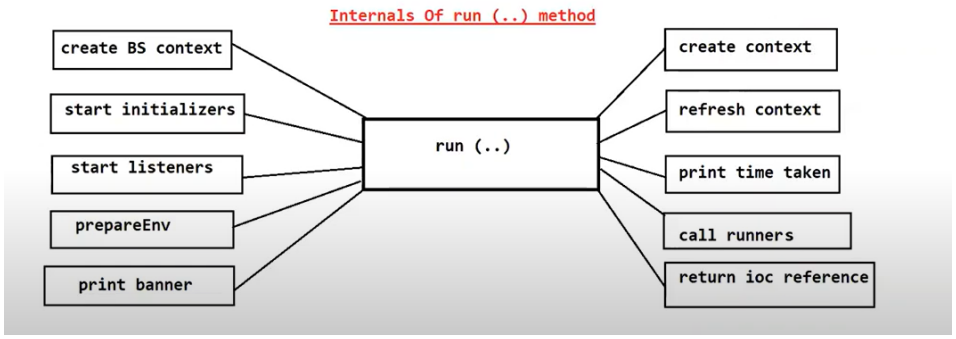
@SpringBootConfiguration

@EnableAutoConfiguration

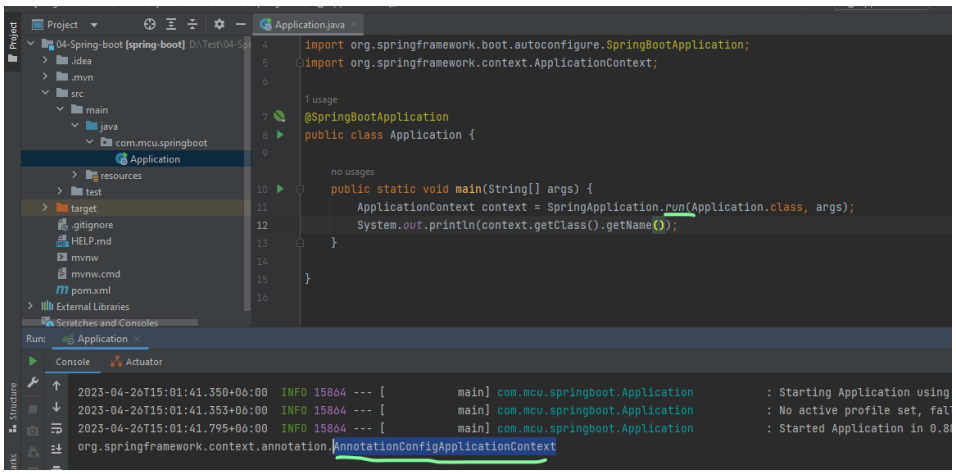
@ComponentScan

➢ **SpringApplication.run (**Application.class, args**)** method **contains bootstrap logic**. It is the entry point for boot application execution

1) **start stop watch**
2) **start listeners**
3) **prepare Environments**
4) **print Banner**
5) **create IOC container**
6) **refresh Container**
7) **stop that stop watch**
8) **print time taken to start application**
9) **call runners**
10) **return IOC reference**



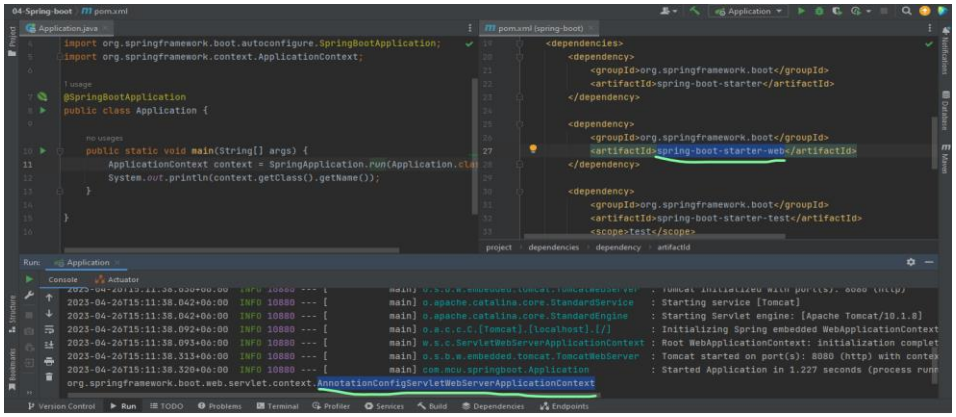
Internal mechanism of run method



If it has only "spring-boot-starter" then Run method internally use this "AnnotationConfigApplicationContext" class to start **IOC** Container.

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

If it has "**spring-boot-starter-web**" then **Run** method internally use this **"AnnotationConfigServletWebServerApplicationContext"** class to start **IOC** Container.



How IOC Container Will Start in Spring Boot?

++
How ioc container will start in Spring Boot ?
++

=> For boot-starter, run( ) method using "AnnotationConfigApplicationContext" class to start IOC container

=> For boot-starter-web, run ( ) method using "AnnotationConfigServletWebServerApplicationContext" to start IOC

=> For starter-webflux, run ( ) method using "AnnotationConfigReactiveWebServerApplicationContext" to start IOC

-> boot-starter is used to create standalone applications

-> boot-starter-web dependency is used to develop web applications

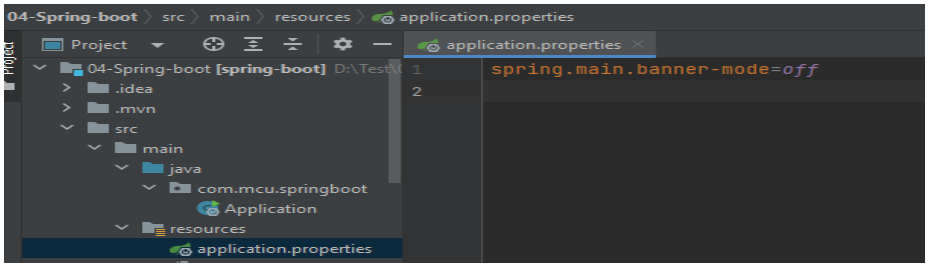
-> boot-starter-webflux is dependency is used to develop applications with Reactive Programming

Runners in Spring Boot

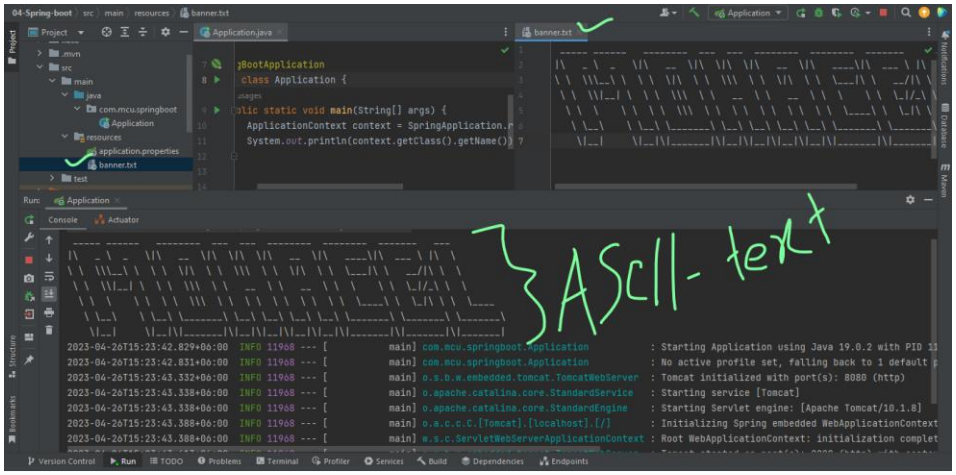
```
++++++++++++++++++++
Runners in Spring Boot
++++++++++++++++++++

=> Runners are getting called at the end of run ( ) method

=> If we want to execute any logic only once when our application got started then we can use Runners concept

=> In Spring Boot we have 2 types of Runners

1) ApplicationRunner
2) CommandLineRunner

=> Both runners are functional interface. They have only one abstract method i.e run (..) methodd


Use cases
----------------

1) send email to management once application started
2) read data from db tables and store into cache memory
```

Banner in Spring Boot

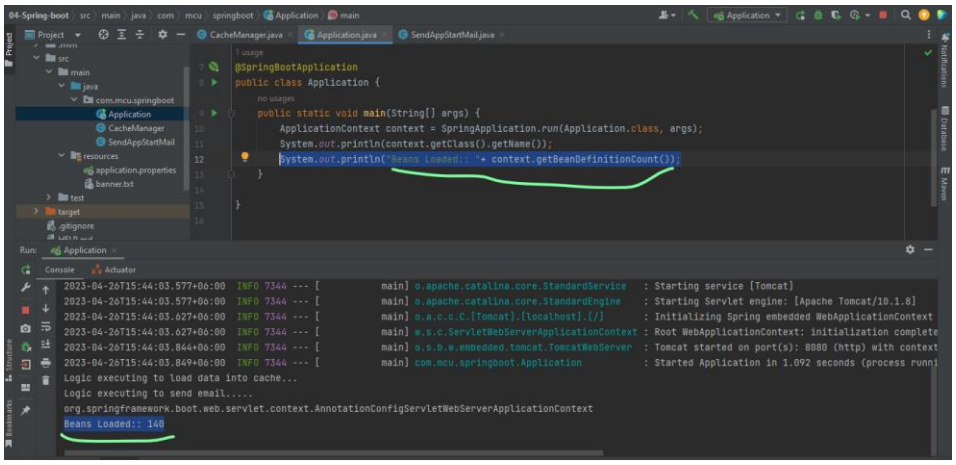
```
++++++++++++++++++
Banner in Spring Boot
++++++++++++++++++

-> In Boot application console we can see spring logo that is called as Banner in Spring Boot
-> We can customize banner text by creating "banner.txt" file under src/main/resources
-> We can keep our company or project name as Banner text in Ascii format
-> Spring Boot banner having below 3 modes

      1) console  ---- > it is default mode
      2) log
      3) off

Note: If we set banner mode as off then banner will not printed
```



It will off the banner mode.



Create a **“banner.txt”** file under resources folder and paste “ASCII-TEXT” it will automatically print the banner



To check how many beans are loaded **“context.getBeanDefinitionCount()”**

What is ComponentScan?

What is ComponentScan ?
++++++++++++++++++++

-> The process of scanning packges in the application to identify spring bean classes is called as Component Scan

-> Component Scan is built-in concept

-> Component Scanning will start from base package

Note: The package which contains start class is called as base package.

-> After base packge scanning completed it will scan sub packages of base package.

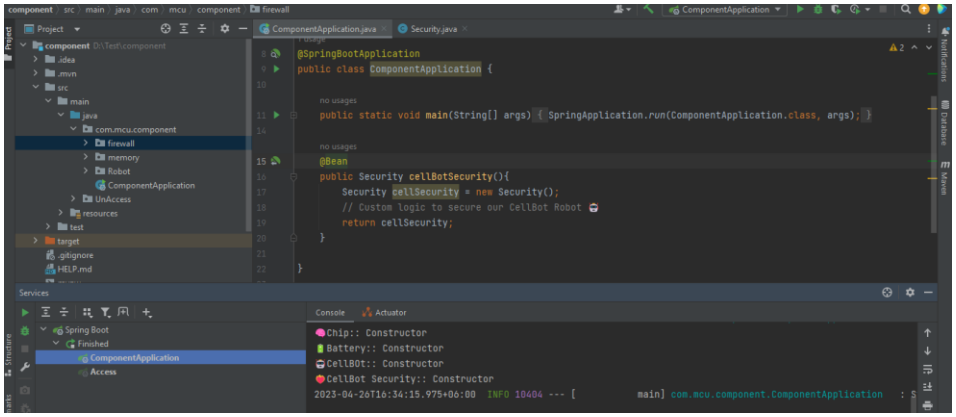
Note: The package names which are starting with base package name are called as sub packages.

com.mcu.component (Base package)

com.mcu.component.memory (Sub-Package)

com.mcu.component.Robot (Sub-Package)

UnAccess. Encryption ---(this is not sub package so ignore it for IOC)



What is @Bean Annotation

```
++++++++++++++++++++++++
What is @Bean annotation
++++++++++++++++++++++++

-> @Bean is a method level annotation

-> When we want to customize any object creation then we will use @Bean annotation for that

@Configuration
public class AppConfig {

    @Bean
    public AppSecurity createInstance() {
        AppSecurity as = new AppSecurity();
        // custom logic to secure our functionality
        return as;
    }
}


Note: @Bean method we can write in any spring bean class but recommended to write in @Configuration class like above
```

How to Represent java class as a spring Bean?

```
How to represent java class as a Spring Bean?
++++++++++++++++++++++++++++++++++++++++

@Component
@Service
@Repository

@Controller
@RestController

@Configuration
@Bean


-> All the above annotations are class level annotations but  @Bean is method level annotation


-> @Component is a general purpose annotation to represent java class as Spring Bean
```

```
-> @Service annotation is a specialization of @Component annotation. This is also used to represent java class as spring
bean. It is allowing for implementation classes to be autodetected through classpath scanning.

    Note: For business classes we will use @Service annotation

-> @ Repository annotation is a specialization of @Component annotaton. This is also used to represent java class as
spring bean. It is having Data Access Exception translation

    Note: For dao classes we will use @Repository
```

-> In Web applications to represent java class as controller we will use @Controller annotation. It is used for C2B communication.

-> In Distributed application to represent java class as distributed component we will use @RestController annotation. It is used for B2B communication.

-> If we want to perform customized configurations then we will use @Configuration annotation along with @Bean methods. Based on requirement we can have multiple @Configuration classes in the project.

Ex: Swagger Config, DB Config, RestTemplate Config, Kafka Config, Redis Config, Security Config etc...

Note: @Bean annotated method we can keep in any spring bean class but it is highly recommended to keep them in @Configuration classes.

Auto-wiring

Autowiring
+++++++++

-> Autowiring is used to perform dependency injection

-> The process of injecting one class object into another class object is called as dependency injection.

-> In Spring framework IOC container will perform dependency injection

-> We will provide instructions to IOC to perform DI in 2 ways

1) Manual Wiring (using ref attribute in beans config xml file)

2) Autowiring

-> Autowiring means IOC will identify dependent object and it will inject into target object

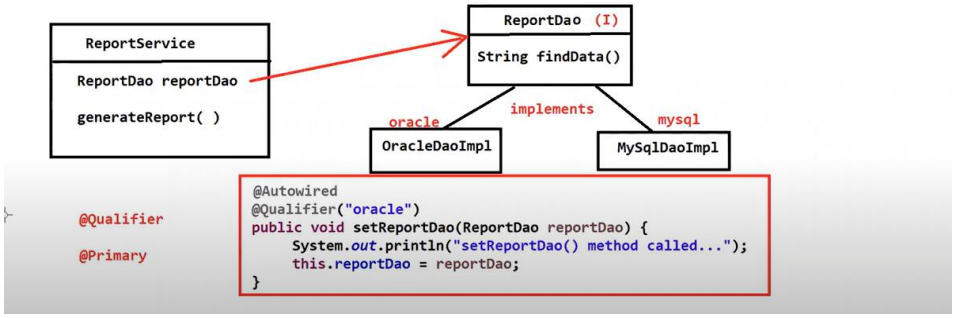
-> Autowiring will use below modes to perform Dependency Injection

1) byName
2) byType
3) constructor (internally it will use byType only)

-> To perform Autowiring we will use @Autowired annotation

-> @Autowired annotation we can use at 3 places in the program

1) variable level (Field injection - FI)
2) setter method level (Setter Injection - SI)
3) constructor (Constructor Injection - CI)



```java
package com.mcu.autowiring.robot;

import com.mcu.autowiring.Battey.Cell;
import com.mcu.autowiring.Battey.ChargeCell;
import com.mcu.autowiring.memeory.Chip;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ZeroTwo {
    //3rd way field/variable injection
//    @Autowired
    private Chip memory;
    //3rd way field/variable injection
//    @Autowired
    private Cell battery;
    private String NeonCell = "com.mcu.autowiring.Battey.NeonCell";
    private String ChargeCell = "com.mcu.autowiring.Battey.ChargeCell";

    public ZeroTwo() {
        System.out.println("ZeroTwo no-args :: Constructor Executed");
    }


    //1st way constructor injection
    //    @Autowired
    public ZeroTwo(Chip memory, ChargeCell battery) {
        this.memory = memory;
        this.battery = battery;
        System.out.println("ZeroTwo args :: Constructor Executed");
    }

    //2nd way by setter injection
    @Autowired
    public void setMemory(Chip memory) {
        this.memory = memory;
    }
    //2nd way by setter injection
    @Autowired
    public void setBattery(Cell battery) {
        this.battery = battery;
    }
    public void Status(){
        System.out.println("HI, I am Zero Two 🤖");
        String status = String.format(memory.chipStatus()+"\n"+ "Battery
status: "+battery.chargeStatus()+"%%");
        System.out.println(status);

        if (battery.getClass().getName().equals(NeonCell)){
            System.out.println(battery.getClass().getName()+" 💚💚💚");
        }else{
            System.out.println(battery.getClass().getName()+" 💔💔💔");
        }
```

```
    }
}
```

Setter Injection vs Constructor Injection vs Field Injection

```
SI vs CI vs FI
++++++++++++

-> Setter Injection will be performed through setter method
-> It is mandatory to specify @Autowired annoation at setter method
-> If we don't specify @Autowired annotation then DI will not happen (Partial Injection)
-> If DI not happend, when we call methods then we will get NullPointerException
-> Target Bean will be created first then setter method will be called to inject dependent


-> Constructor injection will be performed through constructor
-> If we have more than one constructor then we have to specify @Autowired at constructor level
-> If we have only one parameterized constructor then @Autowired is optional
-> In CI, dependent bean will be created first then target bean will be created
-> Partial Injection is not possible in CI


-> FieldInjection will be performed through Reflection API
-> FI violating OOPS principles (private variable getting initialized from outside using Reflecion)
-> It is simple to use
-> Most of the programmers will use FI in projects because it us jus one line code
```

Bean Life Cycle

```
++++++++++++++
Bean Life Cycle
++++++++++++++

-> The java class which represented as Spring Bean is called as Bean Class

-> The java class which is managed by IOC is called as Spring Bean

-> Spring Beans life cycle will be managed by IOC

-> For Spring Beans Obj creation and Obj destruction will be taken by IOC container

-> We can execute life cycle methods for Spring Bean

-> In Spring Boot, we can work with Bean Lifecycle methods in 2 ways

            1) By Implementing interfaces ( IntializingBean & DisposableBean )

            2) By Using Annotations ( @PostConstruct & @PreDestory )
```

```
// interface approach

@Component
public class Motor implements InitializingBean, DisposableBean {

	public Motor() {
		System.out.println("Motor :: Constructor");
	}

	@Override
	public void afterPropertiesSet() throws Exception {
		System.out.println("afterPropertiesSet() method called....");
	}

	@Override
	public void destroy() throws Exception {
		System.out.println("destroy() method called....");
	}
}
```

```
// Annotation Approach

@Component
public class Engine {

	public Engine() {
		System.out.println("Engine :: Constructor");
	}

	@PostConstruct
	public void init() {
		System.out.println("start engine....");
	}

	@PreDestroy
	public void destroy() {
		System.out.println("stop engine...");
	}

}
```

Spring Core Context Questions(video 15)

1) what is Framework
2) Web Framework (Ex: Struts)
3) ORM Framework (Ex: Hibernate)
4) Application Development Framework (Ex: Spring)

5) Spring Framework Advantages
6) Spring Modules
7) What is Spring Core
8) IOC Container
9) Dependency Injection
10) Strategy Design Pattern
11) Spring Application Creation
12) Spring Bean Config File
13) Setter Injection
14) Constructor Injection
15) Field Injection
16) Manual Wiring using “ref” attribute
17) Autowiring (byName, byType)
18) Spring Boot
19) Spring Boot Advantage
20) Spring Boot Application Creation
21) Start Class in Spring Boot
22) @SpringBootApplication Annotation
23) Component Scanning
24) SpringAppliction.run(…) method internals
25) Banner & Banner Mode
26) Runner (Application Runner and Command Line Runner)
27) How IoC creating in Spring Boot (Based on POM starter)
28) Stereotype annotation (@Component, @Service, @Repository)
29) @Configuration, @Bean
30) Bean Scope (singleton & prototype)
31) @Autowired, @Qualifier & @Profile
32) Bean life Cycle
    a. Interface Approach (InitializingBean & DisposableBean)
    b. Annotation Approach (@PostConstruct & @PreDestory)

Spring Data JPA

- It is used to develop persistence logic in our application
- The logic which is responsible to communicate with database is called as Persistence Logic
- Already we have JDBC, Spring JDBC, Hibernate, Spring ORM to develop persistence logic
- If we use JDBC, or Spring JDBC or Hibernate or Spring ORM then we should common logics in all DAO classes to perform CURD Operations
- Data JPA simplified persistence logic development by providing pre-defined interfaces with methods
- Data JPA provided Repository methods to perform CURD operations

Note: If we use Data JPA then we don't need to write logic to perform curd operations because Data JPA will take care of that

Data JPA provided Repository interfaces to perform CURD operations

1) **CrudRepository** (Methods to perform Crud Operation)
2) **JpaRepository** (Method to perform Curd Operations + Pagination + Sorting + QBE)

Entity Class

Entity Class
++++++++++

-> The java class which is mapped with DB table is called as Entity class

-> To map java class with DB table we will use below annotations

@Entity : It represents our class as Entity class. (It is mandatory annotation)

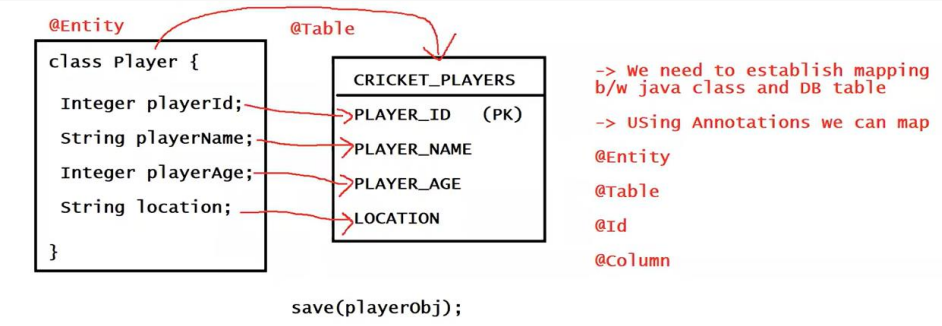
@Table : It is used to map our class with DB Table name

Note : @Table is optional if our class name and table is same. If we don't give @Table then it will consider class name as table name.

@Id : It represents variable mapping with primary column in table (It is mandatory annotation)

@Column : It is used to map our class variables with DB table column names

Note: @Column is optional if class variable name and DB table column names are same. If we don't give @Column then it will consider variable name as column name.



Note: For **Each table** we should **create one Entity class**. Entity **class represents Database** operations should be performed in which **table**.

One Entity class will map with only one table. If you have 500 class then 500 table need. We can't use same class with multiple tables.

Note: For every table we should create one Entity class. Entity class represents DB operations should be performed in which table.

```
@Entity
@Table(name="CRICKET_PLAYERS")
public class Player {

    @Id
    @Column(name="PLAYER_ID")
    private Integer playerId;

    @Column(name="PLAYER_NAME")
    private String playerName;

    @Column(name="PLAYER_AGE")
    private Integer playerAge;

    @Column(name="LOCATION")
    private String location;
}
```

Repository Interface

```
++++++++++++++++++++
Repository Interfaces
++++++++++++++++++++

-> JPA provided repository interfaces to perform Curd Operations

-> For Every DB table we will create one Repository interface by extending Jpa Repository


Syntax:
---------
public interface PlayerRepository extends CrudRepository<Entity, ID>{
}

Example
-----------
public interface PlayerRepository extends CrudRepository<Player, Serializable>{
}
```

Note: When our interface extending properties from JPA Repository interfaces then JPA will provide implementation for our interface in Runtime using Proxy Design Pattern.

Data source Properties

Datasource properties
++++++++++++++++++++

-> Datasource properties represents with which database we want connect

- DB URL
- DB Uname
- DB Pwd
- DB Driver Class

-> We will configure datasource properties in application.properties file or application.yml file

spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/student

spring.datasource.password = admin123

spring.datasource.username = root

ORM Properties

++++++++++++
ORM Properties
+++++++++++++

-> Hibernate provided some additional benefits while developing persistence logic

-> Tables can be created dynamically using "auto-ddl" property

-> We are calling JPA methods to perform DB operations. Those methods will generate queries to execute. To print those queries on console we can use 'show-sql' property

spring.jpa.show-sql = true

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto = create

Building First App using Data JPA

++++++++++++++++++++++++++
Build First App using Data JPA
++++++++++++++++++++++++++

1) Create spring boot application with below dependencies

a) starter-data-jpa
b) mysql-connector

2) Create Entity class using Annotations

3) Create Repository interface by extending CrudRepository

4) Configure DataSource & ORM properties in application.properties file

5) Call Repository methods in start class to perform DB operations

application.properties

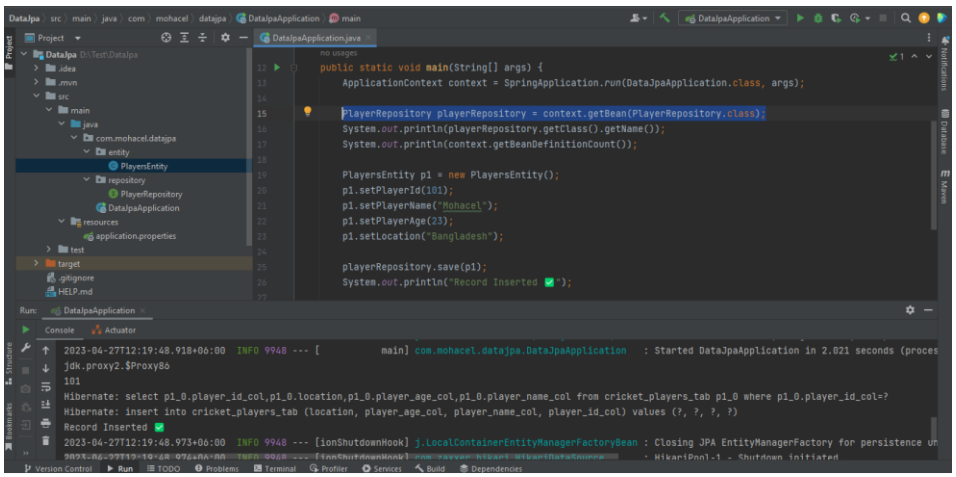
```
# Data Source properties
spring.datasource.driver-class-name  =  com.mysql.cj.jdbc.Driver
spring.datasource.url  =  jdbc:mysql://localhost:3306/sbms
spring.datasource.password =  admin123
spring.datasource.username =  root

# ORM properties
spring.jpa.show-sql  =  true
spring.jpa.properties.hibernate.dialect  =  org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto = update
```

PlayerRepository.java

```
package com.mohacel.datajpa.repository;

import com.mohacel.datajpa.entity.PlayersEntity;
import org.springframework.data.repository.CrudRepository;

3 usages
public interface PlayerRepository extends CrudRepository<PlayersEntity, Integer> {
}
```

PlayersEntity.java

```
@Entity
@Table(name = "CRICKET_PLAYERS_TAB")
public class PlayersEntity {
    3 usages
    @Id
    @Column(name = "PLAYER_ID_COL")
    private Integer playerId;
    3 usages
    @Column(name = "PLAYER_NAME_COL")
    private String playerName;
    3 usages
    @Column(name = "PLAYER_AGE_COL")
    private Integer playerAge;
    3 usages
    private String location;
```



CrudRepository Methods

1) save ( ) :: upsert method (insert and update) - for one record

2) saveAll ( ) :: upsert method (insert and update) - for multiple records

3) findById ( ) :: To retrieve record using primary key

4) findAllById ( ) :: To retrieve multiple records using primary keys

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

6) count ( ) :: To get total records count

7) exists ( ) : To check record presence in table using Primary key

8) deleteById ( ) : To delete single record using primary key

9) deleteAllById ( ) : To delete multiple records using primary keys

10) delete( ) : To delete record using entity object

11) deleteAll ( ) : To delete all records from table

```
package com.mohacel.springbootdatajpa;

import com.mohacel.springbootdatajpa.entity.UserEntity;
import com.mohacel.springbootdatajpa.repository.UserRepository;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```java
import org.springframework.context.ApplicationContext;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

@SpringBootApplication
public class Application {

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

        UserRepository userRepository =
context.getBean(UserRepository.class);

//        <------------------ single record inserted --------------------->
//        UserEntity user1 = new
UserEntity(1002,"Mohacel","Male",23,"Bangladesh");
//        userRepository.save(user1);
//

//        <------------------- multiple record inserted ----------------->
        List<UserEntity> userEntityList = new ArrayList<>();
        userEntityList.add(new
UserEntity(1003,"Adi","Male",23,"Bangladesh"));
        userEntityList.add(new
UserEntity(1004,"Reznu","Male",23,"Bangladesh"));
        userEntityList.add(new
UserEntity(1005,"Asadu","Male",23,"Bangladesh"));
        userEntityList.add(new
UserEntity(1011,"Megh","Female",14,"Bangladesh"));
        userEntityList.add(new
UserEntity(1012,"Fahmida","Female",24,"Bangladesh"));
        userEntityList.add(new
UserEntity(1013,"Aishah","Female",20,"Australia"));
        userEntityList.add(new UserEntity(1014,"ROC","Female",28,"Germany"));

        userRepository.saveAll(userEntityList);
        System.out.println("☑ Record Inserted Successfully into sbms
database (USER_MASTER_TAB) ☑");

//        <------------------  Find By ID  ---------------------->
        Optional<UserEntity > findById = userRepository.findById(1002);
        if(findById.isPresent()){
            System.out.println("☑ "+findById+" ☑");
        }else{
            System.out.println("ID Not Found ✕");
        }

//        <-------------------------  Find ALL By ID ----------------------
------->
        Iterable<UserEntity> allById =
userRepository.findAllById(Arrays.asList(1002, 1003, 1004));
```

```
        System.out.println("-----------------------------------------------
----------------------");
        allById.forEach(i-> System.out.println(i));
        System.out.println("-----------------------------------------------
----------------------");

//        <-------------------------  Find ALL ----------------------------
->
        Iterable<UserEntity> allUser = userRepository.findAll();

System.out.println("🌿🌿🌿
```

UserEntity.java

```java
10 usages
@Entity
@Table(name = "MCU_USER_TAB")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserEntity {
    no usages
    @Id
    @Column(name = "USER_ID_COL")
    private Integer id;
    no usages
    @Column(name = "USER_NAME_COL")
    private String name;
    no usages
    @Column(name = "USER_AGE_COL")
    private Integer age;
    no usages
    @Column(name = "USER_GENDER_COL")
    private String gender;
    no usages
    @Column(name = "USER_COUNTRY_COL")
    private String Country;

}
```

UserEntity.java | IUserRepository.java

```java
package com.mcu.repository;

import com.mcu.entity.UserEntity;
import org.springframework.data.repository.CrudRepository;

3 usages
public interface IUserRepository extends CrudRepository<UserEntity, Integer> {
}
```

```java
UserEntity.java   IUserRepository.java   Application.java

import ...
1 usage
@SpringBootApplication
public class Application {
    no usages
    public static void main(String[] args) {
        ConfigurableApplicationContext Context = SpringApplication.run(Application.class, args);

        // get the bean
        IUserRepository userRepository = Context.getBean(IUserRepository.class);

        // creat a list of user
        List<UserEntity> userList = new ArrayList<>();
        userList.add(new UserEntity( id: 1002, name: "Mohacel", age: 23, gender: "Male", Country: "Bangladesh"));
        userList.add(new UserEntity( id: 1003, name: "Adi", age: 23, gender: "Male", Country: "Bangladesh"));
        userList.add(new UserEntity( id: 1004, name: "Vera", age: 26, gender: "Female", Country: "Philippine"));
        userList.add(new UserEntity( id: 1005, name: "Krystal", age: 24, gender: "Female", Country: "America"));
        userList.add(new UserEntity( id: 1006, name: "Megh", age: 14, gender: "Female", Country: "Bangladesh"));
        userList.add(new UserEntity( id: 1007, name: "Walida", age: 15, gender: "Female", Country: "Bangladesh"));

        // add all user
        userRepository.saveAll(userList);

    }
}
```

```
mysql> use sbms;
Database changed
mysql> show tables;
+---------------------+
| Tables_in_sbms      |
+---------------------+
| cricket_players_tab |
| mcu_user_tab        |
| user_master_tab     |
+---------------------+
3 rows in set (0.00 sec)

mysql> select * from mcu_user_tab;
+-------------+-----------------+--------------+----------------+--------------+
| user_id_col | user_country_col | user_age_col | user_gender_col | user_name_col |
+-------------+-----------------+--------------+----------------+--------------+
|        1002 | Bangladesh      |           23 | Male           | Mohacel      |
|        1003 | Bangladesh      |           23 | Male           | Adi          |
|        1004 | Philippine      |           26 | Female         | Vera         |
|        1005 | America         |           24 | Female         | Krystal      |
|        1006 | Bangladesh      |           14 | Female         | Megh         |
|        1007 | Bangladesh      |           15 | Female         | Walida       |
+-------------+-----------------+--------------+----------------+--------------+
6 rows in set (0.00 sec)
```

Note: In **CrudRepository** we can't use non-primary column to retrieve our data because there is no built-in method for this. But can achieve this by "**findByMethods**" or

"**Custom queries**"

-> In CrudRepository interface we have methods to retrieve records

1) findById (ID) ----> to retrieve based on primary key

2) findAllById(Iterable<ID> ids) ----> to retrieve based on multiple primary keys

3) findAll( ) -----> to retrieve all records

Requirement -1 : Retrieve users records who are belongs to INDIA
Requirement -2 : Retrieve users whose age is below 30 years

Note: user_age and user_country are non-primary columns in the table

-> To retrieve data based on Non-Primary key columns we don't have pre-defined methods. To implement these kind of requirements in Data JPA we have below 2 options

1) findby methods

2) custom queries

Find By Methods

Find By Methods
++++++++++++++

-> Find By Methods are used to construct queries based on method name

-> Method Name is very important to prepare query dynamically

-> Based on our method name, JPA will prepare the query in Runtime and it will execute that query

Syntax : findby+entityClassVariableName (parameters ...)

Note: findBy methods we will write in Repository interface (abstract methods)

Ex: Retrieve user data based on country_name

```
findbycountry(String countryname);
```

Ex : Retrieve user data based on user_age

```
findbyage(Integer age);
```

Note: findbyXXX methods are used for only Retrieval (SELECT) operation.

mysql>

select * from mcu_user_tab where user_country_col='Bangladesh';

```
14    public static void main(String[] args) {
15        ApplicationContext Context = SpringApplication.run(Application.class, args);
16
17        // get the bean
18        IUserRepository userRepository = Context.getBean(IUserRepository.class);
19
20    //...
31
32    //        <------------------------- retrieve data using country --------------------------------->
33        List<UserEntity> byCountry = userRepository.findByCountry("Bangladesh");
34        System.out.println("-------------------------------------------------------------------------------");
35        byCountry.forEach(i-> System.out.println(i));
36        System.out.println("-------------------------------------------------------------------------------");
37
```

```
Run: Application
Console   Actuator
Hibernate: select u1_0.user_id_col,u1_0.user_age_col,u1_0.user_country_col,u1_0.user_gender_col,u1_0.user_name_col f
-------------------------------------------------------------------------------
UserEntity(id=1002, name=Mohacel, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1003, name=Adi, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1006, name=Megh, age=14, gender=Female, country=Bangladesh)
UserEntity(id=1007, name=Walida, age=15, gender=Female, country=Bangladesh)
-------------------------------------------------------------------------------
```

mysql>

select * from mcu_user_tab where user_age_col = 23;

```
14    public static void main(String[] args) {
15        ApplicationContext Context = SpringApplication.run(Application.class, args);
16
17        // get the bean
18        IUserRepository userRepository = Context.getBean(IUserRepository.class);
19
20    //...
37
38
39    //        <------------------------- retrieve data using age --------------------------------->
40        List<UserEntity> byAge = userRepository.findByAge(23);
41        System.out.println("-------------------------------------------------------------------------------");
42        byAge.forEach(i-> System.out.println(i));
43        System.out.println("-------------------------------------------------------------------------------");
44
```

```
Run: Application
Console   Actuator
2023-04-28T10:41:51.789+06:00  INFO 3300 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initializ
2023-04-28T10:41:52.009+06:00  INFO 3300 --- [           main] com.mcu.Application                      : Started A
Hibernate: select u1_0.user_id_col,u1_0.user_age_col,u1_0.user_country_col,u1_0.user_gender_col,u1_0.user_name_col
-------------------------------------------------------------------------------
UserEntity(id=1002, name=Mohacel, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1003, name=Adi, age=23, gender=Male, country=Bangladesh)
-------------------------------------------------------------------------------
```

mysql>

select * from mcu_user_tab where user_age_col >=23;

```
14    public static void main(String[] args) {
15        ApplicationContext Context = SpringApplication.run(Application.class, args);
16
17        // get the bean
18        IUserRepository userRepository = Context.getBean(IUserRepository.class);
19
20    //...
44
45    //        <------------------------- retrieve data using age>=  ---------------------------------->
46        List<UserEntity> byAgeIsGreaterThanEqual = userRepository.findByAgeIsGreaterThanEqual(23);
47        System.out.println("-------------------------------------------------------------------------------");
48        byAgeIsGreaterThanEqual.forEach(i-> System.out.println(i));
49        System.out.println("-------------------------------------------------------------------------------");
50
51
52
```

```
Run: Application
Console   Actuator
--------------------------------------------------------------------------------
UserEntity(id=1002, name=Mohacel, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1003, name=Adi, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1004, name=Vera, age=26, gender=Female, country=Philippine)
UserEntity(id=1005, name=Krystal, age=24, gender=Female, country=America)
--------------------------------------------------------------------------------
```

mysql>

select * from mcu_user_tab where user_country_col in ('Bangladesh', 'America');

```
15    public static void main(String[] args) {
16        ApplicationContext Context = SpringApplication.run(Application.class, args);
17
18        // get the bean
19        IUserRepository userRepository = Context.getBean(IUserRepository.class);
20
21    //...
53        List<UserEntity> byCountries = userRepository.findByCountryIn(Arrays.asList("Bangladesh", "America"));
54        System.out.println("-------------------------------------------------------------------------------");
55        byCountries.forEach(i-> System.out.println(i));
56        System.out.println("-------------------------------------------------------------------------------");
57
```

```
Run: Application
Console   Actuator
2023-04-28T10:51:07.509+06:00  INFO 9136 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
2023-04-28T10:51:07.744+06:00  INFO 9136 --- [           main] com.mcu.Application                      : Started App
Hibernate: select u1_0.user_id_col,u1_0.user_age_col,u1_0.user_country_col,u1_0.user_gender_col,u1_0.user_name_col fr
--------------------------------------------------------------------------------
UserEntity(id=1002, name=Mohacel, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1003, name=Adi, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1005, name=Krystal, age=24, gender=Female, country=America)
UserEntity(id=1006, name=Megh, age=14, gender=Female, country=Bangladesh)
UserEntity(id=1007, name=Walida, age=15, gender=Female, country=Bangladesh)
--------------------------------------------------------------------------------
```

mysql>

select * from mcu_user_tab where user_country_col='Bangladesh' and user_age_col<20;

```
15    public static void main(String[] args) {
16        ApplicationContext Context = SpringApplication.run(Application.class, args);
17
18        // get the bean
19        IUserRepository userRepository = Context.getBean(IUserRepository.class);
20
21    //...
59        List<UserEntity> byCountryAndAge = userRepository.findByCountryAndAgeIsLessThan( country: "Bangladesh",  age: 20);
60        System.out.println("------------------------------------------------------------------------------");
61        byCountryAndAge.forEach(i-> System.out.println(i));
62        System.out.println("------------------------------------------------------------------------------");
```

```
Run:  Application
Console   Actuator
2023-04-28T11:01:56.056+06:00  INFO 10892 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized
2023-04-28T11:01:56.290+06:00  INFO 10892 --- [           main] com.mcu.Application                      : Started Appl
Hibernate: select u1_0.user_id_col,u1_0.user_age_col,u1_0.user_country_col,u1_0.user_gender_col,u1_0.user_name_col from
------------------------------------------------------------------------------
UserEntity(id=1006, name=Megh, age=14, gender=Female, country=Bangladesh)
UserEntity(id=1007, name=Walida, age=15, gender=Female, country=Bangladesh)
------------------------------------------------------------------------------
```

mysql>

select * from mcu_user_tab where user_country_col='America' and user_age_col>20 and user_gender_col='Female';

```
15    public static void main(String[] args) {
16        ApplicationContext Context = SpringApplication.run(Application.class, args);
17
18        // get the bean
19        IUserRepository userRepository = Context.getBean(IUserRepository.class);
20
21    //...
65        List<UserEntity> byCountryAndAgeAndGender = userRepository.findByCountryAndAgeIsGreaterThanAndGender( country: "America", age: 20, gender: "Female");
66        System.out.println("------------------------------------------------------------------------------");
67        byCountryAndAgeAndGender.forEach(i-> System.out.println(i));
68        System.out.println("------------------------------------------------------------------------------");
```

```
Run:  Application
Console   Actuator
Hibernate: select u1_0.user_id_col,u1_0.user_age_col,u1_0.user_country_col,u1_0.user_gender_col,u1_0.user_name_col from mcu_user_tab u1_0 where u1_0.use
------------------------------------------------------------------------------
UserEntity(id=1005, name=Krystal, age=24, gender=Female, country=America)
------------------------------------------------------------------------------
2023-04-28T11:09:17.925+06:00  INFO 14284 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persiste
2023-04-28T11:09:17.927+06:00  INFO 14284 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2023-04-28T11:09:17.931+06:00  INFO 14284 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

findBy Methods Example

Note: JPA will construct the quires at run time, programmer doesn't need to create full method. **Just define abstract method inside repository interface**.

```
++++++++++++++++++++++++++++++++ FindBy Methods Examples+++++++++++++++++++++++++++++++++++++++++++++++++++++++++

public interface UserRepository extends CrudRepository<User, Integer> {

        // select * from user_master where user_country=?;
        public List<User> findByCountry(String cname);

        // select * from user_master where user_age=?;
        public List<User> findByAge(Integer age);

        // select * from user_master where user_age >= ?;
        public List<User> findByAgeGreaterThanEqual(Integer age);

        //select * from user_master where user_country in (?,?,? ...);
        public List<User> findByCountryIn(List<String> countries);

        // select * from user_master where user_country='India' and user_age=25;
        public List<User> findByCountryAndAge(String cname, Integer age);

        // select * from user_master where user_country='India' and user_age=25 and user_gender='Male';
        public List<User> findByCountryAndAgeAndGender(String cname, Integer age, String gender);
}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Custom Queries inside Repository Interface

```
08-Spring Data JPA Custom Quries > src > main > java > com > mcu > repository > IUserRepository > findByAgeIsGreaterThanEqual
IUserRepository.java

3 usages
public interface IUserRepository extends CrudRepository<UserEntity, Integer> {

    //mysql> select * from mcu_user_tab where user_country_col='Bangladesh';
    no usages
    List<UserEntity> findByCountry(String country);

    //mysql> select * from mcu_user_tab where user_age_col = 23;
    no usages
    List<UserEntity> findByAge(Integer age);

    //mysql> select * from mcu_user_tab where user_age_col >=23;
    no usages
    List<UserEntity> findByAgeIsGreaterThanEqual(Integer age);

    //mysql> select * from mcu_user_tab where user_country_col in ('Bangladesh', 'America');
    no usages
    List<UserEntity> findByCountryIn(List<String> country);

    //select * from mcu_user_tab where user_country_col='Bangladesh' and user_age_col<20;
    no usages
    List<UserEntity> findByCountryAndAgeIsLessThan(String country, Integer age);

    //mysql> select * from mcu_user_tab where user_country_col='America' and user_age_col>20 and user_gender_col='Female';
    1 usage
    List<UserEntity> findByCountryAndAgeIsGreaterThanAndGender(String country, Integer age, String gender);
}
```

Custom Queries

++++++++++++++++
Custom Queries
++++++++++++++++

-> We can write our own query and we can execute that query using JPA that is called as Custom Query

-> Custom Queries we can write in 2 ways

1) HQL Queries

2) Native SQL Queries

Native SQL Vs HQL
++++++++++++++++

-> Native SQL queries are DB dependent
-> HQL Queries are DB in-dependent

-> Native Queries will use table name and column names in the query directley
-> HQL queris will use 'Entity Class Name' and 'Entity Class Variable Names' in the query

-> Native SQL queries will execute in Database Directley
-> HQL queries can't execute in DB directley (HQL queries will be converted to SQL query using Dialect class before execution)

-> As SQL queries are executing directley, performance wise they are good. HQL queries should be converted before execution hence HQL queries will take more time than SQL queries.

-> Performance wise SQL queries are good
-> Maintenence wise HQL queies are good

#Retrieve all the record of UserEntity

SQL: *select * from mcu_user_tab;*

HQL: *From UserEntity*

#Retrieve all the record of UserEntity who are belongs to country 'Bangladesh'

SQL: *select * from mcu_user_tab where user_country_col='Bangladesh';*

HQL: *from UserEntity where country='Bangladesh'*

#Retrieve all the record of UserEntity who are belongs to country 'Bangladesh' and age is 23

SQL: *select * from mcu_user_tab where user_country_col='Bangladesh and 'user_age_col = 23;*

HQL: *from UserEntity where country='Bangladesh' and age=23;*

#Retrieve User_Name and User_Age column based on country='Bangladesh';

SQL: *select user_name_col, user_age_col from mcu_user_tab where user_country_col='Bangladesh';*

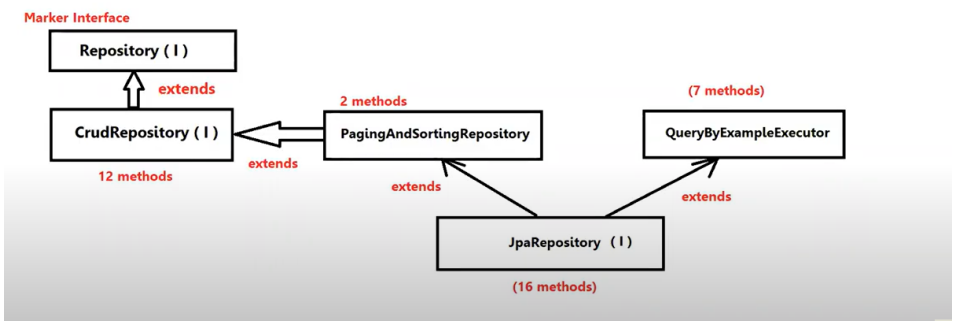
HQL: *select name, age from UserEntity where country='Bangladesh'*

```
3 usages
public interface IUserRepository extends CrudRepository<UserEntity, Integer> {

//...

//    <------------------------------------ Custom queries    ------------------------------------
    1 usage
    @Query(value = "from UserEntity ")
    List<UserEntity> getAllUserHql();

    1 usage
    @Query(value = "from UserEntity where country=:userCountry and gender=:userGender")
    List<UserEntity> getAllUserByCountryAndGenderHql(String userCountry, String userGender);

    1 usage
    @Query(value = "select * from MCU_USER_TAB", nativeQuery = true)
    List<UserEntity> getAllUserSql();

}
```

Dynamic input

```
Application.java
public static void main(String[] args) {
    ApplicationContext Context = SpringApplication.run(Application.class, args);

    // get the bean
    IUserRepository userRepository = Context.getBean(IUserRepository.class);

//...

    // custom queries "HQL"
    System.out.println("------------------------------------  HQL  ------------------------------------");
    List<UserEntity> userList1 = userRepository.getAllUserHql();
    userList1.forEach(i-> System.out.println(i));
    System.out.println("------------------------------------------------------------------------------");
```

```
Run: Application
Console   Actuator
------------------------------------  HQL  ------------------------------------
Hibernate: select u1_0.user_id_col,u1_0.user_age_col,u1_0.user_country_col,u1_0.user_gender_col,u1_0.user_name_col from mcu_user_tab u1_0
UserEntity(id=1002, name=Mohacel, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1003, name=Adi, age=23, gender=Male, country=Bangladesh)
UserEntity(id=1004, name=Vera, age=26, gender=Female, country=Philippine)
UserEntity(id=1005, name=Krystal, age=24, gender=Female, country=America)
UserEntity(id=1006, name=Megh, age=14, gender=Female, country=Bangladesh)
UserEntity(id=1007, name=Walida, age=15, gender=Female, country=Bangladesh)
------------------------------------------------------------------------------
```



JpaRepository

JpaRepository
++++++++++++++++++

-> It is a predefined interface available in data jpa

-> Using JpaRepository interface also we can perform CRUD operations with DB tables

-> JpaRepository also having several methods to perform CRUD operations

-> JpaRepository having support for Pagination + Sorting + QBE (Query By Example)

Note: CrudRepository interface doesn't support Pagination + Sorting + QBE

```java
package com.mohacel.jparepository.entity;

import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

import java.time.LocalDate;
@Data
@NoArgsConstructor
@Entity
@Table(name = "CLUB_MEMBER_TAB")
public class ClubMemberEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "MEMBER_ID_COL")
    private Integer memberId;
    @Column(name = "MEMBER_NAME_COL")
    private String name;
    @Column(name = "MEMBER_AGE_COL")
    private Integer age;
    @Column(name = "MEMBER_GENDER_COL")
    private String gender;

    @CreationTimestamp
    @Column(name = "JOINING_DATE_COL", updatable = false)
    private LocalDate joiningDate;

    @UpdateTimestamp
    @Column(name = "UPDATE_DATE_COL", insertable = false)
    private LocalDate updatedDate;

    public ClubMemberEntity(String name, Integer age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
```

ClubMemberEntity.java

```java
package com.mohacel.jparepository.repository;

import com.mohacel.jparepository.entity.ClubMemberEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface IMemberRepository extends JpaRepository<ClubMemberEntity,
Integer> {
}
```

IMemberRepository.java

```
package com.mohacel.jparepository;

import com.mohacel.jparepository.entity.ClubMemberEntity;
import com.mohacel.jparepository.repository.IMemberRepository;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.data.domain.Sort;

import java.util.Arrays;
import java.util.List;

@SpringBootApplication
public class Application {

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

        ClubMemberEntity member1 = new ClubMemberEntity("Mohacel",23,"Male");
        ClubMemberEntity member2 = new ClubMemberEntity("Walida",
15,"Female");
        ClubMemberEntity member3 = new ClubMemberEntity("Aisha",
21,"Female");
        ClubMemberEntity member4 = new ClubMemberEntity("Krystal",
27,"Female");
        ClubMemberEntity member5 = new ClubMemberEntity("Tushar", 19,"Male");
        ClubMemberEntity member6 = new ClubMemberEntity("Adi", 23,"Male");
        ClubMemberEntity member7 = new ClubMemberEntity("Asadu", 23,"Male");
        ClubMemberEntity member8 = new ClubMemberEntity("Reznu", 23,"Male");
        ClubMemberEntity member9 = new ClubMemberEntity("Rayhan", 24,"Male");
        ClubMemberEntity member10 = new ClubMemberEntity("Takumi",
20,"Male");

memberRepository.saveAll(Arrays.asList(member1,member2,member3,member4,member
5,member6,member7,member8,member9,member10));


        System.out.println("-------------------------------------------------
----------------------------");
        List<ClubMemberEntity> members =
memberRepository.findAll(Sort.by("memberId").descending());
        members.forEach(i-> System.out.println(i));
        System.out.println("-------------------------------------------------
----------------------------");


    }
```

```
}
```

Application.java

```
mysql> select * from  club_member_tab;
+---------------+---------------+-------------------+-----------------+----------------+-----------------+
| member_id_col | member_age_col | member_gender_col | joining_date_col | member_name_col | update_date_col |
+---------------+---------------+-------------------+-----------------+----------------+-----------------+
|             1 |            23 | Male              | 2023-04-29       | Mohacel        | NULL            |
|             2 |            15 | Female            | 2023-04-29       | Walida         | NULL            |
|             3 |            21 | Female            | 2023-04-29       | Aisha          | NULL            |
|             4 |            27 | Female            | 2023-04-29       | Krystal        | NULL            |
|             5 |            19 | Male              | 2023-04-29       | Tushar         | NULL            |
|             6 |            23 | Male              | 2023-04-29       | Adi            | NULL            |
|             7 |            23 | Male              | 2023-04-29       | Asadu          | NULL            |
|             8 |            23 | Male              | 2023-04-29       | Reznu          | NULL            |
|             9 |            24 | Male              | 2023-04-29       | Rayhan         | NULL            |
|            10 |            20 | Male              | 2023-04-29       | Takumi         | NULL            |
+---------------+---------------+-------------------+-----------------+----------------+-----------------+
10 rows in set (0.00 sec)
```

```
2023-04-29T11:48:17.836+06:00  INFO 11704 --- [           main] com.mohacel.jparepository.Application    : Started Application in
--------------------------------------------------------------------------------
Hibernate: select c1_0.member_id_col,c1_0.member_age_col,c1_0.member_gender_col,c1_0.joining_date_col,c1_0.member_name_col,c1_0.up
ClubMemberEntity(memberId=10, name=Takumi, age=20, gender=Male, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=9, name=Rayhan, age=24, gender=Male, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=8, name=Reznu, age=23, gender=Male, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=7, name=Asadu, age=23, gender=Male, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=6, name=Adi, age=23, gender=Male, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=5, name=Tushar, age=19, gender=Male, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=4, name=Krystal, age=27, gender=Female, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=3, name=Aisha, age=21, gender=Female, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=2, name=Walida, age=15, gender=Female, joiningDate=2023-04-29, updatedDate=null)
ClubMemberEntity(memberId=1, name=Mohacel, age=23, gender=Male, joiningDate=2023-04-29, updatedDate=null)
--------------------------------------------------------------------------------
2023-04-29T11:48:17.927+06:00  INFO 11704 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManag
```

Pagination

++++++++++
Pagination
++++++++++

-> The process of dividing all the records into multiple pages is called as Pagination

-> If we retrieve all the records at once then performance issues we will get in the application

-> When we have lot of data in table to display then we will divide those records into multiple pages and we will display in front-end

-> Data will be displayed based on below 2 conditions

=> PAGE NUMBER (User landed on which page)

=> PAGE SIZE (How many records should be displayed in single page)

-> Data will be displayed based on below 2 conditions

=> PAGE NUMBER (User landed on which page)

=> PAGE SIZE (How many records should be displayed in single page)

```
int pageSize = 3;
int pageNo  = 1;   (it will come from UI)

PageRequest pageRequest = PageRequest.of(pageNo-1, pageSize);

Page<User> pageData = repository.findAll (pageRequest) ;

int totalPages = pageData.getTotalPages();
System.out.println("Total Pages :: "+ totalPages);

List<User> users = pageData.getContent();
users.forEach(user -> {
        System.out.println(user);
});
```

Query By Example (QBE)

++++++++++++++++++++++
Query By Example (QBE)
++++++++++++++++++++

-> It is used to prepare the query dynamically

-> To implement Dynamic Search Option we can use QBE concept

```
User entity  = new User();

entity.setCountry("India");
entity.setAge(25);

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

List<User> users = repository.findAll(example);

users.forEach(user -> {
        System.out.println(user);
});
```

```java
//      <---------------------------------------- Sorting  ---------------------------------------------->

        System.out.println("--------------------------------------------------------------------------------");
        List<ClubMemberEntity> membersList1 = memberRepository.findAll(Sort.by( ...properties: "memberId").descending());
        membersList1.forEach(i-> System.out.println(i));
        System.out.println("--------------------------------------------------------------------------------");

        System.out.println("--------------------------------------------------------------------------------");
        List<ClubMemberEntity> membersList2 = memberRepository.findAll(Sort.by( ...properties: "name").ascending().and(Sort.by( ...properties: "age").ascending()));
        membersList1.forEach(i-> System.out.println(i));
        System.out.println("--------------------------------------------------------------------------------");
```

Sorting

```java
//      <-----------------------------   pagination    ------------------------------------------->
        int pageNumber = 0;
        int sizePerPage = 4;
        System.out.println("--------------------------------------------------------------------------------");
        Page<ClubMemberEntity> all = memberRepository.findAll(PageRequest.of(pageNumber, sizePerPage));
        all.forEach(i-> System.out.println(i));
        System.out.println("--------------------------------------------------------------------------------");
        System.out.println("Total Page Number:: "+all.getTotalPages());
```

Pagination

```java
//      <--------------------------------Query By Example ---------------------------------------->

        ClubMemberEntity memberEntity = new ClubMemberEntity();
        memberEntity.setGender("Male");
        memberEntity.setAge(23);

        Example<ClubMemberEntity> example = Example.of(memberEntity);
        List<ClubMemberEntity> allMembers = memberRepository.findAll(example);
        System.out.println("--------------------------------------------------------------------------------");
        allMembers.forEach(i-> System.out.println(i));
        System.out.println("--------------------------------------------------------------------------------");
```

Query by Example (QBE)

```java
//      <----------------------- Example and Sort --------------------------------->
        ClubMemberEntity memberEntity = new ClubMemberEntity();
        memberEntity.setGender("Male");
        System.out.println("--------------------------------  Example and  Sort----------------------------------------");
        List<ClubMemberEntity> allMembers = memberRepository.findAll(Example.of(memberEntity), Sort.by( ...properties: "name").descending());
        allMembers.forEach(i-> System.out.println(i));
        System.out.println("--------------------------------------------------------------------------------");


        System.out.println("--------------------------------- Example and Page ----------------------------------------");
        Page<ClubMemberEntity> memberEntities = memberRepository.findAll(Example.of(memberEntity), PageRequest.of( page: 1,  size: 5));
        memberEntities.forEach(i-> System.out.println(i));
        System.out.println("--------------------------------------------------------------------------------");
```

Example and (Sort/ page)

Timestamping in Data JPA

++++++++++++++++++++++++
Timestamping in Data JPA
++++++++++++++++++++++++

-> For every table we need to maintain below 4 columns to analyze data

CREATED_DATE
CREATED_BY

UPDATED_DATE
UPDATED_BY

Note: In realtime we will maintain these 4 columns for every table

-> CREATED_BY & UPDATED_BY represents which user creating and updating records in table. In Web applications we will use logged in username and we will set for these 2 columns.

-> CREATED_DATE & UPDATED_DATE columns represents when record got created and when record got updated.

-> Instead of setting CREATED_DATE & UPDATE_DATE columns values manually we can below annotations

1) @CreationTimestamp

2) @UpdateTimestamp

```java
public class ClubMemberEntity {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "MEMBER_ID_COL")
    private Integer memberId;
    1 usage
    @Column(name = "MEMBER_NAME_COL")
    private String name;
    1 usage
    @Column(name = "MEMBER_AGE_COL")
    private Integer age;
    1 usage
    @Column(name = "MEMBER_GENDER_COL")
    private String gender;
    no usages
    @CreationTimestamp
    @Column(name = "JOINING_DATE_COL", updatable = false)
    private LocalDate joiningDate;
    no usages
    @UpdateTimestamp
    @Column(name = "UPDATE_DATE_COL", insertable = false)
    private LocalDate updatedDate;
```

Timestamping

-> updatable = false means that column value should not updated when we do update operation on the table.

-> insertable = false means that column value should not inserted when we do insert operation on the table.

Primary Keys

++++++++++++
Primary Keys
++++++++++++

-> Primary Key is a constraint (rule)

-> Primary constraint is the combination of below 2 constraints

1) UNIQUE

2) NOT NULL

-> When we use PRIMARY KEY constraint for a column then that column value shouldn't be null and it should be unique.

-> It is not recommended to set values for Primary Key columns manually

-> We will use Generators to generate the value for primary key column

```
public class ClubMemberEntity {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "MEMBER_ID_COL")
    private Integer memberId;

    1 usage
    @Column(name = "MEMBER_NAME_COL")
    private String name;
    1 usage
    @Column(name = "MEMBER_AGE_COL")
    private Integer age;
    1 usage
    @Column(name = "MEMBER_GENDER_COL")
    private String gender;
    no usages
    @CreationTimestamp
    @Column(name = "JOINING_DATE_COL", updatable = false)
    private LocalDate joiningDate;
    no usages
    @UpdateTimestamp
    @Column(name = "UPDATE_DATE_COL", insertable = false)
    private LocalDate updatedDate;
```

-> To use Generator we will specify @GeneratedValue annotation

-> In MYSQL Database we will use auto_increment to generate value for primary key. To specify auto_increment we will use IDENTITY generator.

```
@Id
@Column(name = "PRODUCT_ID")
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer pid;
```

-> In Oracle database we will use sequence concept to generate value for primary key column. For every primary key one sequence will be created in database

Custom primary key Generator

Step1: create a class that ***implements IdentifierGenerator***

```java
package com.mohacel.customkeys.generator;

import org.hibernate.engine.spi.SharedSessionContractImplementor;
import org.hibernate.id.IdentifierGenerator;
import java.io.Serializable;

public class EmpIdGenerator implements IdentifierGenerator {

    @Override
    public Serializable generate(SharedSessionContractImplementor session,
Object object) {
        String prefix = "dev";
        String query = "SELECT MAX(EMP_ID_COL) FROM EMPLOYEE_TAB";
        String maxId = (String)
session.createNativeQuery(query).uniqueResult();

        if (maxId != null && maxId.matches(prefix + "\\d+")) {
            int id = Integer.parseInt(maxId.substring(prefix.length()));
            return prefix + (++id);
        }

        return prefix + "1001";
    }
}
```

Step2: *override the "**generate**" method* with prefix

Step2.1: in query use primary column name table name

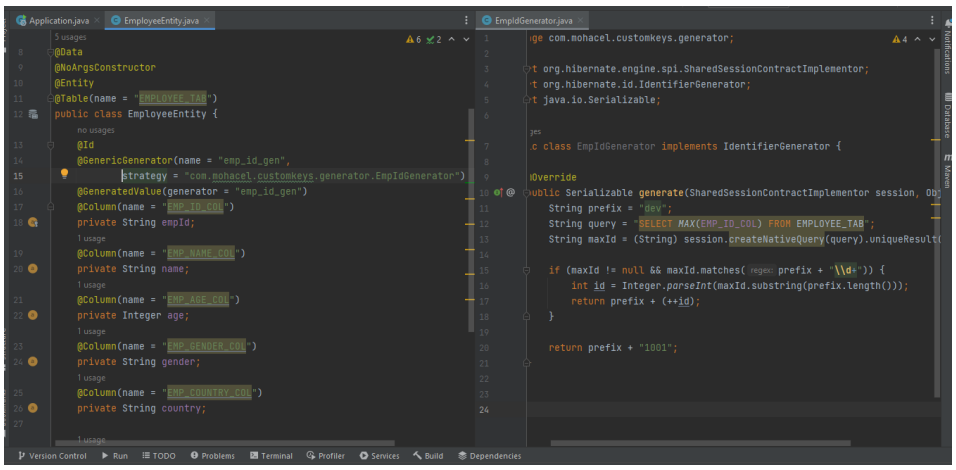
Step3: use **"@GenericGenerator"** as id in entity class and add the implemented class**.**

```java
no usages
@Id
@GenericGenerator(name = "emp_id_gen", strategy = "com.mohacel.customkeys.generator.EmpIdGenerator")
@GeneratedValue(generator = "emp_id_gen")
@Column(name = "EMP_ID_COL")
private String empId;
```

```java
package com.mohacel.customkeys.entity;

import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

@Data
```

```java
@NoArgsConstructor
@Entity
@Table(name = "EMPLOYEE_TAB")
public class EmployeeEntity {
    @Id
    @GenericGenerator(name = "emp_id_gen", strategy =
"com.mohacel.customkeys.generator.EmpIdGenerator")
    @GeneratedValue(generator = "emp_id_gen")
    @Column(name = "EMP_ID_COL")
    private String empId;
    @Column(name = "EMP_NAME_COL")
    private String name;
    @Column(name = "EMP_AGE_COL")
    private Integer age;
    @Column(name = "EMP_GENDER_COL")
    private String gender;
    @Column(name = "EMP_COUNTRY_COL")
    private String country;

    public EmployeeEntity(String name, Integer age, String gender, String
country) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.country = country;
    }
}
```

```java
1 usage
@SpringBootApplication
public class Application {

    no usages
    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);

        IEmployeeRepository employeeRepository = context.getBean(IEmployeeRepository.class);
        EmployeeEntity employee = new EmployeeEntity( name: "Mohacel", age: 23, gender: "Male", country: "Bangladesh");
        employeeRepository.save(employee);

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


    }

}
```



Custom primary key

Composite Key with Custom key

Step1: Create an entity class where two "**@Id**" annotation is used for two different keys.

```
package com.mohacel.entity;

import com.mohacel.customKey.EmpCompositeKey;
import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name = "EMPLOYEE_TAB")
@Data
@NoArgsConstructor
@AllArgsConstructor
@IdClass(EmpCompositeKey.class)
public class EmployeeEntity {
    @Id
    @Column(name = "EMP_ID_COL")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer empId;
```

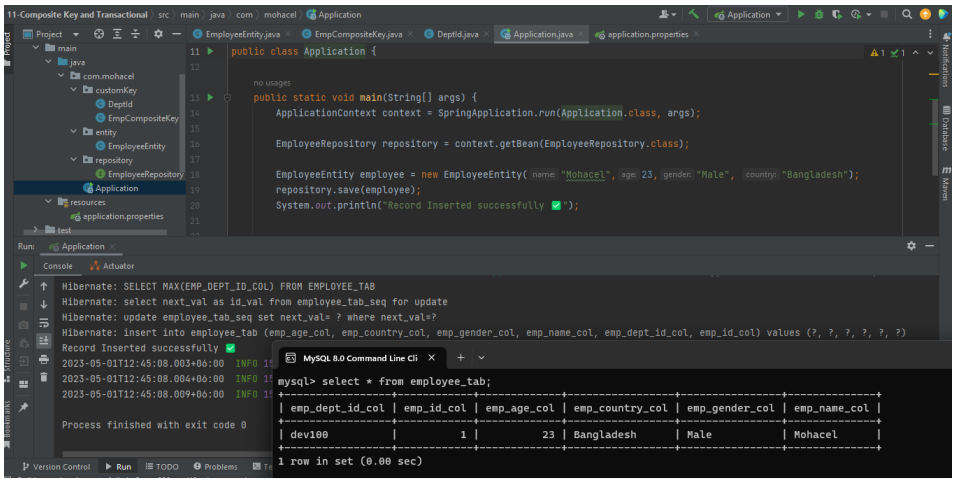
```
    @Id
    @Column(name = "EMP_DEPT_ID_COL")
    @GenericGenerator(name = "emp_id_gen", strategy =
"com.mohacel.customKey.DeptId")
    @GeneratedValue(generator = "emp_id_gen")

    private String deptId;

    @Column(name = "EMP_NAME_COL")
    private String name;
    @Column(name = "EMP_AGE_COL")
    private Integer age;
    @Column(name = "EMP_GENDER_COL")
    private String gender;
    @Column(name = "EMP_COUNTRY_COL")
    private String country;

    public EmployeeEntity( String name, Integer age, String gender, String
country) {

        this.name = name;
        this.age = age;
        this.gender = gender;
        this.country = country;
    }
}
```

Step2: create a separate class with these 2 primary fields and this class must implement "**Serializable**".

```
package com.mohacel.customKey;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;


import java.io.Serializable;
@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
public class EmpCompositeKey implements Serializable {
    private Integer empId;
    private String deptId;
}
```

Step3: create your custom key class that implement "**IdentifierGenerator**" and must override "**generate**" method.

```java
package com.mohacel.customKey;

import org.hibernate.HibernateException;
import org.hibernate.engine.spi.SharedSessionContractImplementor;
import org.hibernate.id.IdentifierGenerator;

import java.io.Serializable;

public class DeptId implements IdentifierGenerator {


    @Override
    public Serializable generate(SharedSessionContractImplementor session,
Object o) throws HibernateException {
        String prefix = "dev";
        String query = "SELECT MAX(EMP_DEPT_ID_COL) FROM EMPLOYEE_TAB";
        String maxId =(String)
session.createNativeQuery(query).uniqueResult();
        if(maxId != null && maxId.matches(prefix+"\\d+")){
            Integer id = Integer.parseInt(maxId.substring(prefix.length()));
            return   prefix +(++id);
        }
        return prefix+"100";
    }
}
```



Transaction Management in Data JPA

-> Unit amount of work is called as Transaction

-> When we are performing non select operations (insert/update/delete) we need to deal with Transactions

-> For select operations transaction is not required (Retrieval)

-> In Data JPA it is care of transaction managment

-> If all operations are successful then we should commit transaction

-> If any operation is failed in the transaction then we need to rollback that transaction

-> When we are working with transactions we need to ACID properties

A - Atomicity

C - Consistence

I - Isolation

D - Durability

```
@Transactional(rollbackFor = Exception.class)
public void transfer(){
    double amount = 450.00;
    SourceAccount  sender = new SourceAccount("Mohacel",amount);
    DestinationAccount receiver = new DestinationAccount("Krystal", amount);


    source.save(sender);
    int CustomError = 5/0;
    destination.save(receiver);
    System.out.println("Transaction Successful ☑");
}
```

Unit amount of work that perform by a method is a transaction, if there have any exception occurred then this transaction must be rollback mean no data will be save.

Insert image into MySQL Database

Step1: Create byte[ ] field for image into your entity class

```
package com.mohacel.img.entity;

import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@Entity
@Table(name = "USER_TABLE")
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer userId;
    private String email;
    private String name;
    private Integer age;
    private String gender;
    @Lob
    private byte[] image;

    public UserEntity(String email, String name, Integer age, String gender)
{
        this.email = email;
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
```

Step2: Create repository class

```
package com.mohacel.img.repository;

import com.mohacel.img.entity.UserEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface IUserRepository extends JpaRepository<UserEntity, Integer> {
}
```

Step3: Create a service class where you can define a method to save data into database.

```
package com.mohacel.img.service;

import com.mohacel.img.entity.UserEntity;
import com.mohacel.img.repository.IUserRepository;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.io.File;
import java.io.FileInputStream;
import java.nio.file.Files;
import java.nio.file.Paths;

@Service
public class UserService  {
    @Autowired
    private IUserRepository repository;

    public void save() throws Exception{

        UserEntity user = new
UserEntity("mohacel.hosen06@gmail.com","Mofacel",23,"Male");


        String imgPath = "D:\\Test\\mohacel.jpg";
        long size = Files.size(Paths.get(imgPath));
        byte[] img =  new byte[(int) size];
        FileInputStream fis = new FileInputStream( new File(imgPath));
        fis.read(img);
        user.setImage(img);
        fis.close();

        repository.save(user);
    }

}
```

Step4: to save imge—

a) Read the imge path
b) Read the image size (it will help to default[4GB] size to imageSize)
c) Create **file input stream** with **File object** and pass the imgPath as File's parameter
d) Read and store them into imgArray
e) Set the img and close the file input stream

```yaml
spring.datasource.driver-class-name  =  com.mysql.cj.jdbc.Driver
spring.datasource.password =  admin123
spring.datasource.username =  root
spring.datasource.url=jdbc:mysql://localhost:3306/sbms?sessionVariables=sql_m
ode='NO_ENGINE_SUBSTITUTION'&jdbcCompliantTruncation=false

spring.jpa.show-sql  =  true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto = create
```

Note:: if column size is too large then add this into **application.properties**
spring.datasource.url=jdbc:mysql://localhost:3306/sbms?sessionVariables=sql_mode='NO_ENGINE_SUBSTITUTION'&jdbcCompliantTruncation=false

1) `String imgPath = "D:\\Test\\mohacel.jpg";`: This line defines the path of the image file that you want to read. In this case, the image file is located at `"D:\\Test\\mohacel.jpg"`.

2) `long size = Files.size(Paths.get(imgPath));`: This line retrieves the size of the image file in bytes using the `Files.size()` method. It takes the image file path (`imgPath`) as input and returns the size of the file in bytes. The size is stored in the `size` variable.

3) `byte[] img = new byte[(int) size];`: This line creates a byte array named `img` with a size equal to the size of the image file. Since the `size` variable is of type `long`, it needs to be cast to an `int` to create the byte array.

4) `FileInputStream fis = new FileInputStream(new File(imgPath));`: This line creates a new `FileInputStream` object named `fis` to read the contents of the image file. It takes the image file path (`imgPath`) as input and wraps it in a `File` object before passing it to the `FileInputStream` constructor.

5) `fis.read(img);`: This line reads the content of the image file into the `img` byte array using the `read()` method of the `FileInputStream` class. The `read()` method reads up to `img.length` bytes from the input stream (`fis`) and stores them in the `img` array.

6) `user.setImage(img);`: This line assigns the `img` byte array to the `image` property of the `user` object. It assumes that there is a `setImage()` method in the `User` class that accepts a byte array as a parameter and sets it as the image data for the user.

7) `fis.close();`: This line closes the `FileInputStream` to release any system resources associated with it.

In summary, the code reads the content of an image file located at the specified path into a byte array, and then assigns that byte array to the `image` property of a `user` object.

Data JPA Questions (31 videos)

1) What is Data JPA
2) Advantages of Data JPA
3) Entity class with annotations
4) Repository interface
5) CrudRepository
6) JpaRepository
7) CrudRepository interface methods
8) JpaRepository interface methods
9) Pagination
8) Sorting
9) Query By Example
10) findByXXX methods
11) Custom Queries
12) Native SQL vs HQL
13) Dialect class
14) Timestamping
15) Generators
16) Primary Keys
17) Composite Primary keys
18) Tx Rollback
19) Large Object (@Lob)

Spring Web MVC

++++++++++++++
Spring Web MVC
++++++++++++++

-> Spring Web MVC is one module available in the spring framework

-> Using Spring Web MVC module we can develop 2 types of applications

1) Web Applications

2) Distributed Application

-> Web Applications will have user interface (UI)
-> Customers can access web applications directley using internet
-> Web Applications meant for customer to business communication (C 2 B)

Ex: facebook, gmail, linkedin, naukri etc...

-> Distributed Applications are meant for Business to Business Communication (B 2 B)
-> If one application is communicating with another application then we call them as Distributed apps

Advantage of Spring MVC

++++++++++++++++++++++++++++
Advantages of Spring Web MVC
++++++++++++++++++++++++++++

1) Easily we can develop web & distributed applications using Web MVC module

2) It supports Multiple Presentation Technologies (JSP & Thymeleaf)

3) I 18 N Support (Internationalization)

4) Form Bindings (Form Data will be binded to Object and vice versa)

5) Form Tag Library (To simplify forms development with Dynamic behvaiour)

6) Having support for XML to Java object conversion and vice versa

7) Having support for JSON to java object conversion and vice versa

Architecture of Spring Web MVC

++++++++++++++++++++++++++
Spring Web MVC Architecture
++++++++++++++++++++++++++

1) Dispatcher Servlet (Front Controller)
2) Handler Mapper
3) Controller
4) Model And View
5) ViewResolver
6) View

-> DispatcherServlet is a pre-defined servlet class in Spring Web MVC
-> DispatcherServlet is called as Front Controller / Framework Servlet
-> It is responsible to perform pre-processing and post-processing for every request

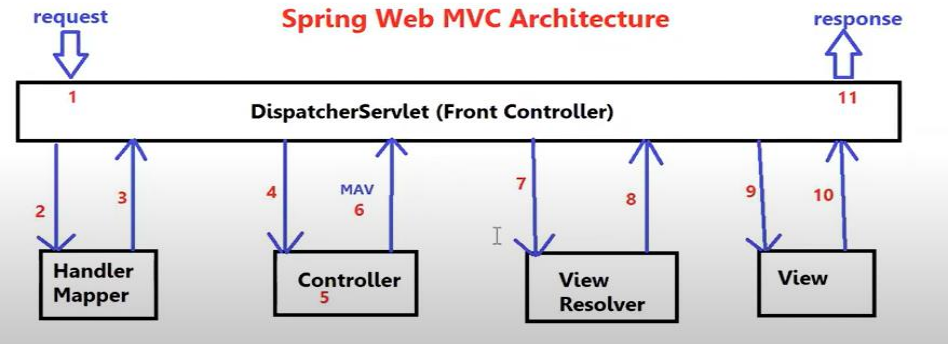
-> HandlerMapper is a pre-defined class in spring web mvc
-> HandlerMapper is used to identify Request Handler
-> It will identify which request should be processed by which Controller class
-> HandlerMapper will identify Request Handler based on URL Pattern

-> Controller is a class which contains logic to handle request and response
-> Controller is also called as Request Handler
-> We will create Controller classes using @Controller annotation
-> Controller will return data to Dispatcher in ModelAndView object

-> Model represents Data in Key-Value pair format
-> View represents presentation logical file name
-> To display data in view file we will use ModelAndView object

-> ViewResolver is used to identify where view files available in our project
-> ViewResolver is responsible to identify physical location of view files

-> View component is used to render model data on physical view file to display that to end user



Building First Web Application using Spring Web MVC

++
Building First Web Application using Spring Web MVC
++

1) Create Spring Starter application with below dependencies

- **a) spring-boot-starter-web**
- **b) tomcat-embed-jasper**
- **c) devtools**

2) Create Controller class and write Required methods

3) Create View Files with Presentation logic

4) Configure ViewResolver in application.properties file

5) Run the application and test it.

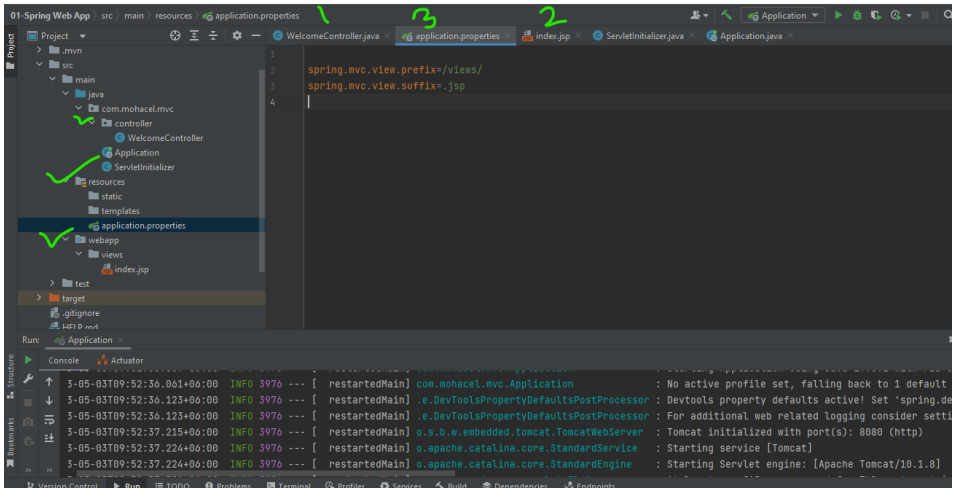
Note-1 : web-starter will provide the support to build web apps with MVC architecture and it provides Tomcat as default embedded container (we no neeed to setup server manually).

Note-2 : tomcat-embed-jasper will provide the support to work with JSP files in Spring Web MVC

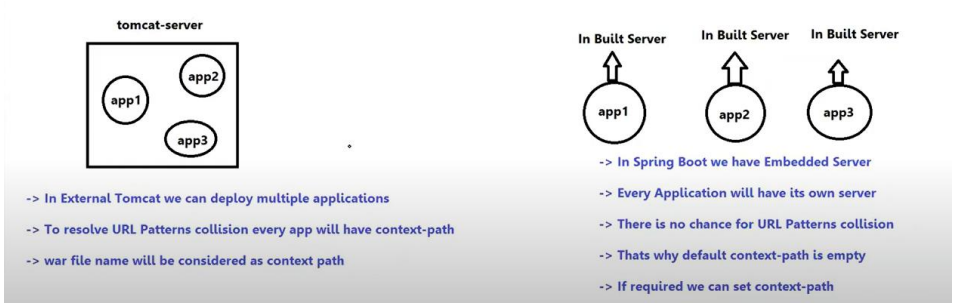
Note-3 : devtools is used to re-start the server when changes happend in the code.

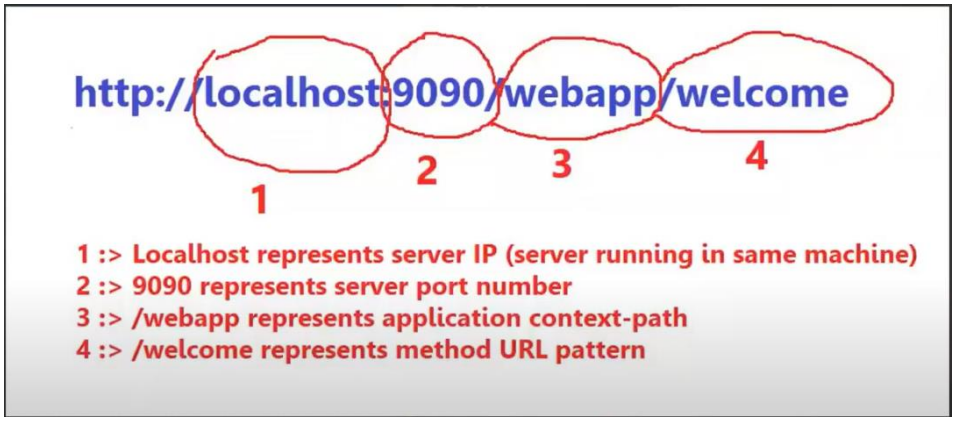
Note-4 : Java class will be represented as a Spring Controller using @Controller annotation

Note-5 : Controller class methods should be binded to HTTP Protocol methods to handle HTTP Requests




```
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp
```





What is Context-Path?

++++++++++++++++++++
What is Context-Path ?
++++++++++++++++++++

-> Context-Path represents name of our application

-> In Spring Boot, the default context-path is empty

-> In Spring Boot we can set our own context-path using below property in application.properties file

server.servlet.context-path=/webapp

-> When we set context we have to access our application using context-path in URL.

URL : http://localhost:8080/webapp/welcome

FirstController.java | welcome.jsp | application.properties

```
spring.mvc.view.prefix=/pages/
spring.mvc.view.suffix=.jsp


server.servlet.context-path=/webapp

server.port=9090

```

Sending Data from Controller to UI

We can send the data from controller to UI in 3 ways---

1) **ModelAndView**
2) **Model**
3) **@ResponseBody**

Approach-1

```
@GetMapping("/welcome")
public ModelAndView welcomeMsg() {
    ModelAndView mav = new ModelAndView();
    mav.addObject("msg", "Welcome to Ashok IT");
    mav.setViewName("welcome");  //settging view name
    return mav;
}
```

Approach-2

```
@GetMapping("/greet")
public String getGreetMsg(Model model) {
    String msgTxt = "Good Morning..";
    model.addAttribute("msg", msgTxt);
    return "greet";  // returning view name
}
```

Approach-3

```
@GetMapping("/wish")
@ResponseBody
public String getWishMsg() {
    String msg = "All the best...!!!";
    return msg;  //returning direct response
}
```

@RestController = @Controller + @ResponseBody

++++++++++++++++++++++++++++++++++++++

Forms Development Using Spring Web MVC

++++++++++++++++++++++++++++++++++++++

-> Forms are very important in every web application

-> Forms are used to collect data from the user

Ex: Login form, Registration form, Search Forms etc....

-> Spring Web MVC provided form tag library to develop forms easily

-> Spring Form Tag Library contains several Tags

<form:form >
<form:input >
<form:password>
<form:radioButton> & <form:radioButtons>
<form:select>
<form:option> & <form:options>
<form:checkbox> & <form:checkboxes>
<form:hidden>
<form:error>

-> Spring Web MVC support Form Binding that means it can bind form data to object and vice versa.

Note: In servlets we use request.getParameter("key") to capture form data.

-> In Spring Web MVC we no need to use request.getParameter("") to capture form data bcz Web MVC having Form Binding Support.

-> To achieve Form Binding we need to create a binding class
(class variables will be mapped with form fields)

Form Based Application

++++++++++++++++++++++++++++++++++
Steps to build first form based application
++++++++++++++++++++++++++++++++++

1) create boot app with below dependencies

a) web-starter
b) devtools
c) lombok

2) Create Form Binding class

3) Create a controller class with required methods

a) method to display empty form (GET Request Method)
b) method to handle form submission (POST Request Method)

4) Create View Page with presentation logic

To work with spring form we need add this tag

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

RESTful Services

RESTFul Services
++++++++++++++++

-> REST stands for 'Representational State Transfer'

-> RESTFul services are used to develop Distributed Applications with Intereoperability

-> If one application is communicating with another application then those are called as 'Distributed Apps'

-> Intereoperability means irrespective of the platform and language applications can communicate

java app <----------> .Net app

.Net app <----------> Python app

Python app <---------> Java App

-> Distributed applications are used for B2B communications

-> B2B means Business to Business Communication

-> Distributed Application will re-use services of one application in another application

-> RESTful Services are providing 'Intereoperability'

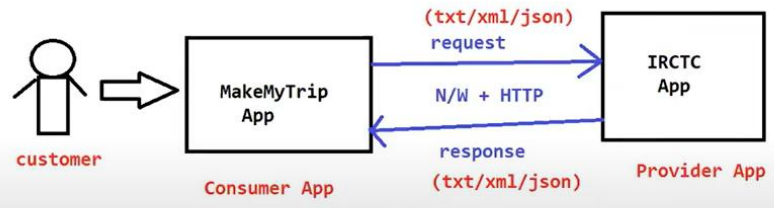
-> Two Actors will be involved in Distributed Applications development

1) Provider
2) Consumer

-> The application which is providing business services is called as Provider Application

-> The application which is accessing business services is called as Consumer Application

Note: One provider application can have multiple consumer applications



-> Provider and Consumer will communicate using HTTP as a mediator

-> Provider and Consumer will exchange the data in Text / XML / JSON format

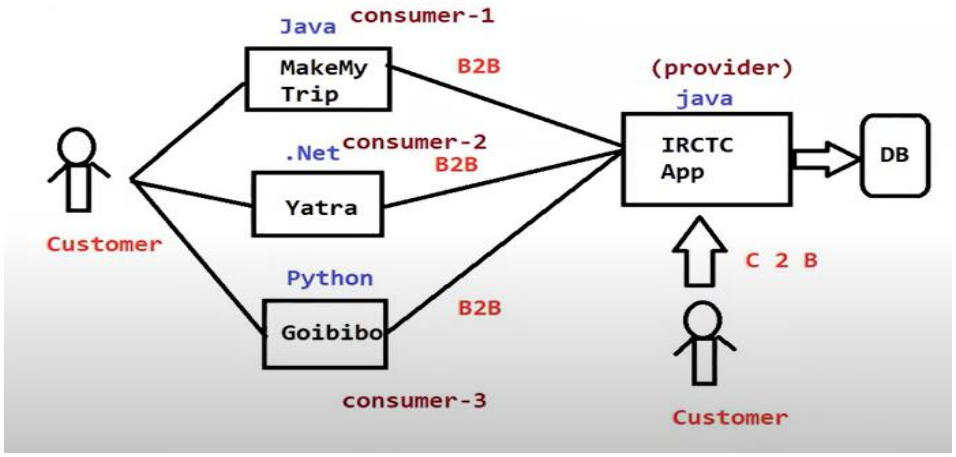
Note: In Industry we will use JSON format to exchange data from one application to another application

-> To start our journey with RESTFul services development we should be good in below areas

1) HTTP Protocol (Methods, Status Codes, Req structure & Res Structure)

2) XML and JAX-B Api

3) JSON and Jackson Api



What is HTTP

```
################
What is HTTP ?
################
```

-> HTTP stands for Hyper Text Transfer Protocol

-> It acts as a mediator between client & server (Consumer & Provider)

-> Consumer application will send HTTP Req to Provider application

-> Provider application will process the request and it will send HTTP response to Consumer

HTTP Methods

###############
HTTP Methods
###############

-> HTTP methods are used to send request from Consumer application to Provider application

-> HTTP method will represent what type of operation client / consumer wants to perform with Provider

GET
POST
PUT
DELETE

-> GET request is used to retrieve data from Server / Provider application.
-> GET request will not have body to send data in the request
-> To send any data using GET request then we will use Path Params & Query Params

Ex: https://www.youtube.com/watch?v=VO818de8sdk

Note: Path Params & Query Params data will be displayed in the URL

Note: It is not recommended to send sensitive / secret data using Path Params & Query Params

-> GET request is Idempotent (means if you send same GET request for multiple times also nothing will change at server)

-> POST request is used to create a new record at server
-> When consumer wants to send huge data/ sensitive data then Consumer will use POST request
-> POST request contains request body
-> POST request is Non-Idempotent

Note: In POST request we can send in URL and in Request Body.

Note: Request Body is the recommended approach to send sensitive data to server

-> PUT request is used to update a record at server
-> When consumer wants to update a record at then consumer will send PUT request to Provider
-> POST request contains request body
-> POST request is Idempotent

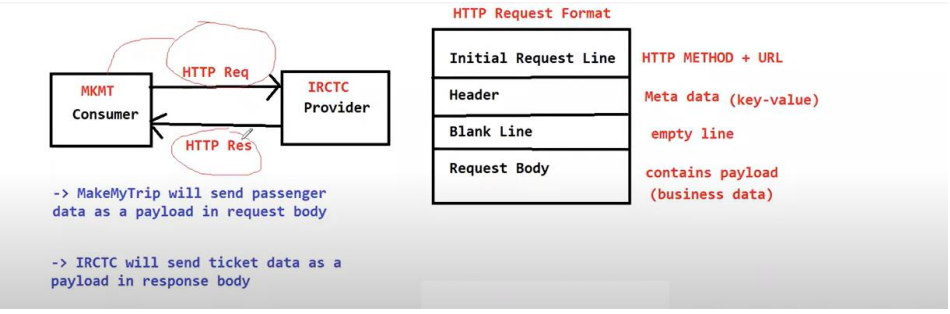
Note: In PUT request we can send in URL and in Request Body.

Note: Request Body is the recommended approach to send sensitive data to server

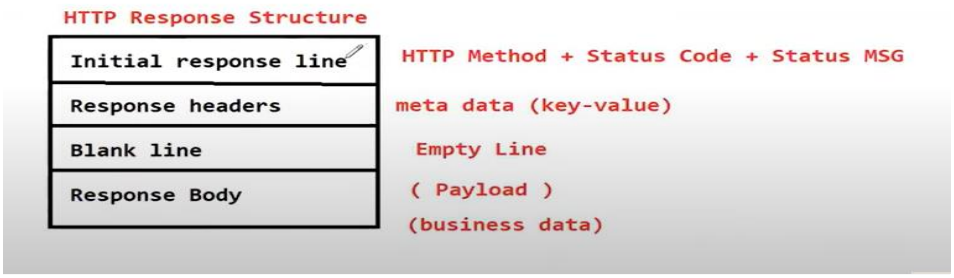
-> DELETE request is used to delete a record at server
-> DELETE request contains request body
-> DELETE request is Idempotent

Note: In DELETE request we can send data in URL and in Request Body.

HTTP Request



HTTP Response



HTTP Request & Response Structure

```
#####################
HTTP Request Structure
#####################

1) Intial Request Line  ( HTTP method + URL )
2) Request Headers ( key-value )
3) Blank Line to seperate Header & Body
4) Request Body (Request Payload)


#####################
HTTP Response Structure
#####################

1) Initial Response line
2) Response Headers (Key-value)
3) Blank Line to seperate Header & Body
4) Response Body (Response Payload)
```

HTTP Status Code

```
##################
HTTP Status Codes
##################

-> HTTP Status codes will represent the status of our request & response

1xx (100 - 199)  ---> INFO

2xx  (200 - 299)  ---> OK (success)

3xx  (300 - 399)  ---> Redirect

4xx (400 - 499) ---> Client Error

5xx (500 - 599) ---> Server Error
```

XML And JAX-B

```
XML & JAX-B
############

-> XML stands for Extensible Markup Language

-> XML is free & open source

-> XML is intereoperable (Language independent & Platform indenpendent)

-> XML  we can use to transfer data from one application to another application

-> XML introduced by w3c org

-> The initial version of xml is 1.0 and the current version of xml is also 1.0

-> XML will represent data in the form of elements

-> An element is the combination of start tag and end tag

      Ex:   <name> Ashok IT </name>
```

We will have 2 types of elements in XML-

1) Simple Element (it contains data directly)
2) Compound Element (it contains child also)

Simple Element: It contain data directly.

```
<name>Mohacel</name>
<id>1002</id>
<department>CSE</department>
```

Compound Element: It contain child data also.

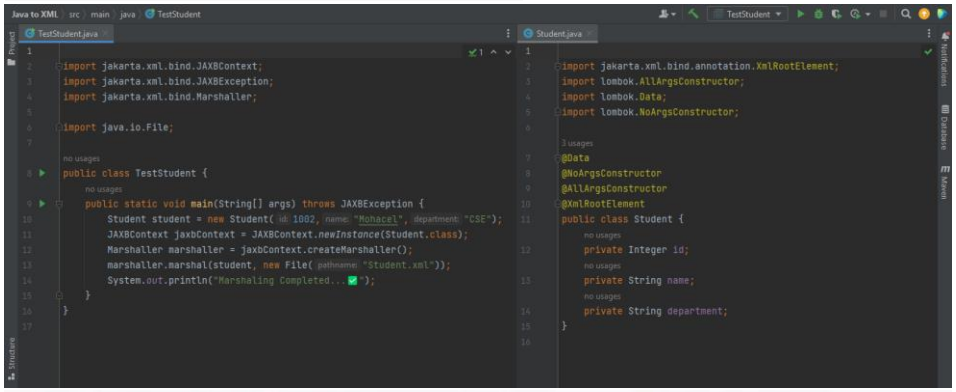
```
<student>
    <name>Mohacel</name>
    <id>1002</id>
    <department>CSE</department>
</student>
```

JAX-B

```
###########
JAX-B
###########

-> JAX-B stands for Java Architecture For XML Binding

-> JAX-B is used to convert Java object to xml and xml to java object

-> JAX-B is free and open source

-> JAX-B given by sun microsystem

-> JAX-B is part of JDK upto 1.8v

-> If you are using JDK 1.8+ version of java then you need to add JAX-B dependency in pom.xml file
```

```
-> The process of converting Java Object into xml is called as "Marshalling"

-> The process of converting XML data to Java Object is called as "Un-Marshalling"

-> To perform Marshalling and Un-Marshalling We need to design Binding Classes.

-> The java class which represents the structure of XML is called as Binding class.

-> JAX-B provided annotations to represent java class as Binding Class.

Note: Binding Class creation is one time operation.

Note: Earlier people used to create Binding Classes using XSD. XSD represents structure of xml.
```

Marshaling and Un-Marshaling



```
public class ConvertXmlToJava {

    public static void main(String[] args) throws Exception {

        File xmlfile = new File("Person.xml");

        JAXBContext context = JAXBContext.newInstance(Person.class);

        Unmarshaller unmarshaller = context.createUnmarshaller();

        Object object = unmarshaller.unmarshal(xmlfile);

        Person person = (Person) object;

        System.out.println(person);
    }
}
```

JSON

```
########
JSON
########
```

-> JSON stands for Java Script object notation

-> JSON will represent data in key-value format

Ex :

```
{
   "id" : 101,
   "name: "raju",
   "age" : 20
}
```

-> JSON is intereoperable (language in-dependent & platform independent)

-> JSON is light weight

-> JSON is both human readable and machine readable format

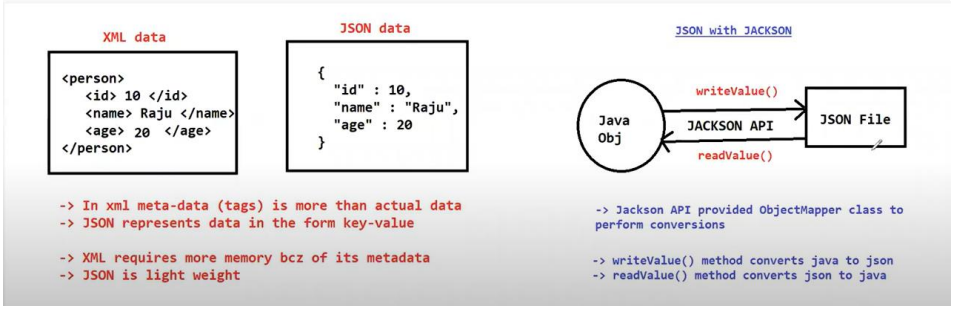
-> In today's world people are using JSON format to exchange the data in B2B communications

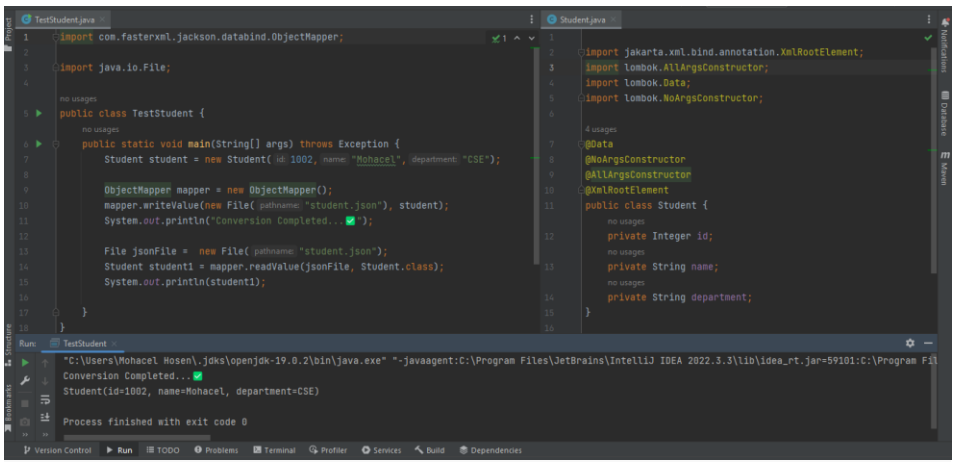
-> Now a days JSON is having more demand than XML because of its simplicity and light weight

-> XML represents data in tags format (open tag & closed tag)
-> Meta data will be more than actual data in XML
-> XML occupies more memory to represent data

-> JSON will take less memory
-> JSON is light weight

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-
databind -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.0</version>
</dependency>
```






```java
public class TestStudent {
    public static void main(String[] args) throws Exception {
        Student student = new Student(1002,"Mohacel","CSE");

        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(new File("student.json"), student);
        System.out.println("Conversion Completed...");

        File jsonFile =  new File("student.json");
        Student student1 = mapper.readValue(jsonFile, Student.class);
        System.out.println(student1);

    }
}
```

How to develop REST API using Java?

++++++++++++++++++++++++++++++++++
How to develop REST API using Java
++++++++++++++++++++++++++++++++++

-> To develop RESFul Services/ REST APIs using java SUN Microsystem released 'JAX-RS' API

-> JAX-RS api having 2 implementations

1) Jersey (Sun Microsystems)
2) REST Easy (JBOSS)

Note: We can develop RESTFul Services using any one of the above implementation

-> Spring framework also provided support to develop RESTFul Services using 'Spring Web MVC' module.

RESTFul Services Architecture
++++++++++++++++++++++++

-> We will have 2 actors in RESTful services

1) Provider / Resource

2) Consumer / Client

-> The application which is providing services to other applications is called as Provider or Resource application

-> The application which is accessing services from other applications is called as Consumer or Client application

-> Client application and Resource application will exchange data in intereoperable format (like XML & JSON)

request
client app <----------> resource app
response

Note: **REST API** are used for B2B communication, there is no Presentation logic

Develop first REST API using spring boot

1) Create Spring starter application with below dependencies

a) web-starter
b) devtools

2) Create RestController with Required methods

Note: To represent java class as Rest Controller we will use @RestController annotation

@RestController = @Controller + @ResponseBody

Note: Every RestController method should be binded to HTTP Protocol method

Ex: @GetMapping, @PostMapping, @PutMapping & @DeleteMapping

3) Run the application and test it.

Note: To test REST APIs we will use POSTMAN tool (It is free)

Note: Download postman tool to test our REST API functionality

Student.java

```
package com.mohacel.path_query.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

9 usages
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {
    no usages
    private Integer id;
    no usages
    private String name;
    no usages
    private String department;
    no usages
    private StudentInterest interest;
}
```

StudentInterest.java

```
package com.mohacel.path_query.model;

import lombok.AllArgsConstructor;
import lombok.Data;

6 usages
@Data
@AllArgsConstructor
public class StudentInterest {
    no usages
    private String hobby;
    no usages
    private String language;
    no usages
    private String project;
}
```

```java
@Controller
public class StudentController {
    @GetMapping("/student")
    public ResponseEntity<Student> getStudent(){
        Student student = new Student(1002,"Mohacel", "CSE",new
StudentInterest("Football","Java","Academic"));
        return new ResponseEntity<>(student, HttpStatus.OK);
    }

    @GetMapping("/all")
    public ResponseEntity<List<Student>> getAll(){
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student(1002, "Mohacel", "CSE",new
StudentInterest("Football","Java","Academic")));
        studentList.add(new Student(1003, "Adi", "CSE",new
StudentInterest("COD","Python","Academic")));
        studentList.add(new Student(1004, "Reznu", "CSE",new
StudentInterest("Football","CPP","Academic")));

        return new ResponseEntity<>(studentList, HttpStatus.OK);
    }
}
```

localhost:8080/all

Save

GET localhost:8080/all Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results Status: 200 OK Time: 87 ms Size: 512 B Save Response

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": 1002,
        "name": "Mohacel",
        "department": "CSE",
        "interest": {
            "hobby": "Football",
            "language": "Java",
            "project": "Academic"
        }
    },
    {
        "id": 1003,
        "name": "Adi",
        "department": "CSE",
        "interest": {
            "hobby": "COD",
            "language": "Python",
            "project": "Academic"
        }
    },
```

Query Params

-> Query Params are used to send data to server in URL directly

-> Query Params will represent data in key-value format

-> Query Params will start with '?'

-> Query Parameters will be seperated by '&'

-> Query Parameters should present only at the end of the URL

Ex: www.ashokit.in/courses?name=SBMS&trainer=Ashok

-> To read Query Parameters from the URL we will use @RequestParam annotation

ProductController.java

```java
import com.mohacel.path_query.model.Product;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

no usages
@Controller
public class ProductController {
    no usages
    @GetMapping("/product")
    public ResponseEntity<Product> productInfo(@RequestParam("id") Integer id,
                                               @RequestParam("name") String name,
                                               @RequestParam("price") Double price){
        Product product = new Product();
        product.setProductId(id);
        product.setProductName(name);
        product.setProductPrice(price);
        return  new ResponseEntity<>(product, HttpStatus.OK);
    }
}
```

Product.java

```java
package com.mohacel.path_query.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

5 usages
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Product {
    no usages
    private Integer productId;
    no usages
    private String productName;
    no usages
    private Double productPrice;
}
```

localhost:8080/product?id=1002&name=Keyboard&price=250.30

Save

GET localhost:8080/product?id=1002&name=Keyboard&price=250.30 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results Status: 200 OK Time: 6 ms Size: 228 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "productId": 1002,
    "productName": "Keyboard",
    "productPrice": 250.3
}
```

Path Parameter or URL Variable

-> Path Parameters are also used to send data to server in URL

-> Path Params will represent data directley in URL (no keys)

-> Path Params can present anywhere in the URL

-> Path Params will be seperated by / (slash)

-> Path Params should be represented in Method URL pattern (Template Pattern)

Example:

*www.mcu.club/department/**{departmentName}**/member/**{memberName}***

```
@Controller
public class ClubMemberController {
    @GetMapping("/club/department/{departmentName}/member/{memberName}")
    public ResponseEntity<ClubMember>
getMember(@PathVariable("departmentName") String departmentName,
@PathVariable("memberName") String memberName ){
        ClubMember member= new ClubMember();
        member.setMemberId(1002);
        member.setMemberName(memberName);
        member.setDepartmentName(departmentName);
        return new ResponseEntity<>(member, HttpStatus.OK);
    }
}
```

ClubMemberController.java

```
import com.mohacel.path_query.model.ClubMember;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

no usages
@Controller
public class ClubMemberController {
    no usages
    @GetMapping("/club/department/{departmentName}/member/{memberName}")
    public ResponseEntity<ClubMember> getMember(@PathVariable("departmentName") String departmentName,
                                                @PathVariable("memberName") String memberName ){
        ClubMember member= new ClubMember();
        member.setMemberId(1002);
        member.setMemberName(memberName);
        member.setDepartmentName(departmentName);
        return new ResponseEntity<>(member, HttpStatus.OK);
    }
}
```

ClubMember.java

```
package com.mohacel.path_query.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

4 usages
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ClubMember {
    no usages
    private Integer memberId;
    no usages
    private String departmentName;
    no usages
    private String memberName;
}
```

http://localhost:8080/club/department/CSE/member/mohacel

GET http://localhost:8080/club/department/CSE/member/mohacel Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results Status: 200 OK Time: 89 ms Size: 227 B Save Response

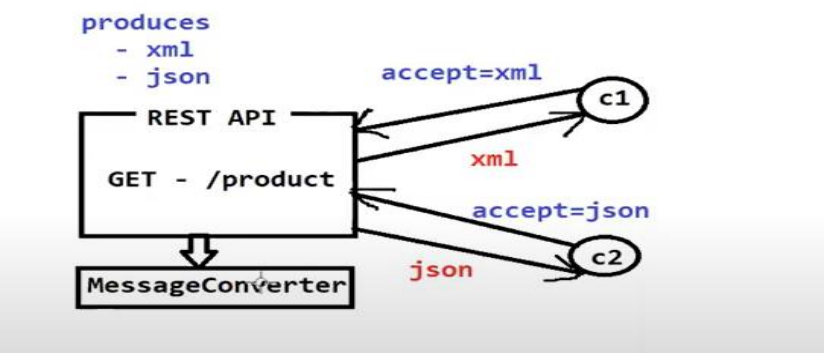
Pretty Raw Preview Visualize JSON

```
{
    "memberId": 1002,
    "departmentName": "CSE",
    "memberName": "mohacel"
}
```

++
Q) When to use Path Params & Query Params ?
++

-> To retrieve more than one record/resource we will use Query Params (filtering)

-> To retreive specific/unique record we will use Path Params (single)



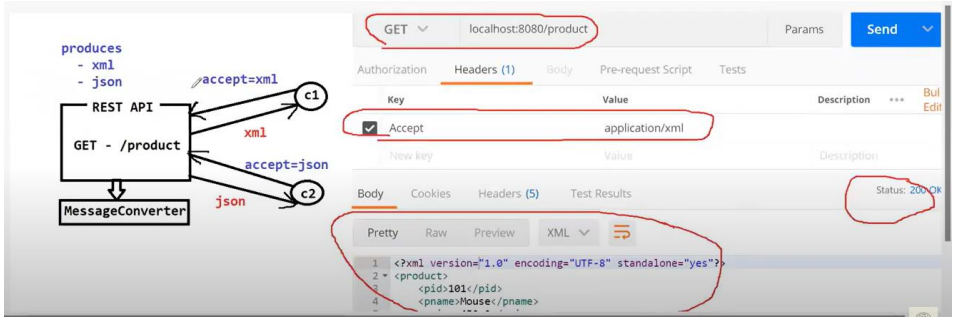
What is Produces

###############
What is Produces
################

-> "produces" is a media type
-> It represents the response formats supported by REST Controller Method
-> One Rest Controller method can support multiple response formats (xml and json)

-> Client should send a request with "Accept" http header
-> Accept header represents in which format client expecting response
-> Based on Accept header value Message Converter will convert the response into client expected format

```
@Controller
public class ProductController {
    @GetMapping(value = "/product", produces = {"application/xml",
"application/json"})
    public ResponseEntity<Product> productInfo(){
        Product product =new Product(101,"RAM",4500.00);
        return  new ResponseEntity<>(product, HttpStatus.OK);
    }
}
```



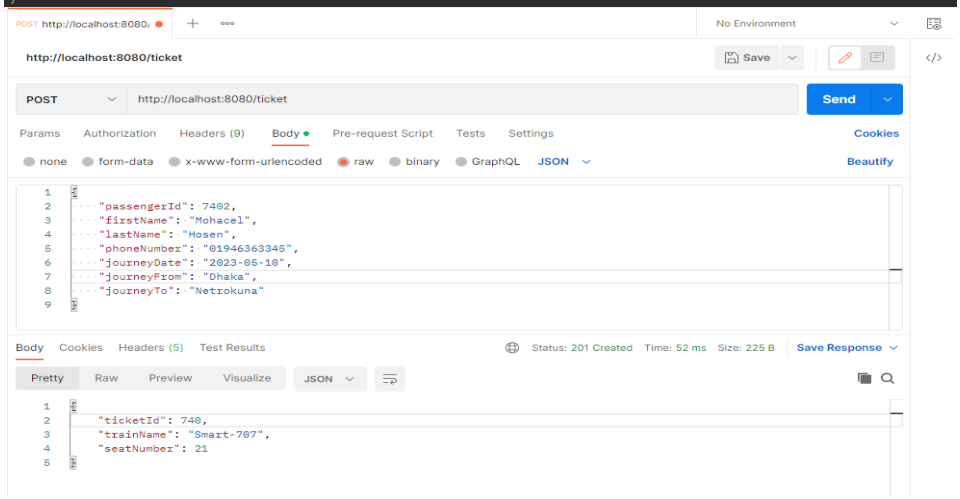
Working with HTTP Post

```
##############################
Working with HTTP POST Request
##############################

-> HTTP POST request is used to create new resource/record at server

-> POST request contains request body

-> Client can send data to server in Request Body

-> To bind Rest Controller method to POST request we willl use @PostMapping

-> To read data from Requet body we will use @RequestBody annotation

-> "consumes" represents in which formats method can take input

-> "Content-Type" header represents in which format client sending data in request body.
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PassengerInfo {
    private Integer passengerId;
    private String firstName;
    private String lastName;
    private String phoneNumber;
    private LocalDate journeyDate;
    private String journeyFrom;
    private String journeyTo;

}
```

```
public class TicketInfo {
    private Integer ticketId;
    private String trainName;
    private Integer seatNumber;

}
```

```java
@Controller
public class TicketController {

    @PostMapping(
            value = "/ticket",
            consumes = {"application/json"},
            produces = {"application/json"}
    )
    public ResponseEntity<TicketInfo>  bookTicket(@RequestBody PassengerInfo
passengerInfo){
        System.out.println(passengerInfo);
        // logic to book ticket
        System.out.println("Passenger Data Stored Successfully...✅");

        TicketInfo ticketStatus = new TicketInfo();
        ticketStatus.setTicketId(740);
        ticketStatus.setTrainName("Smart-707");
        ticketStatus.setSeatNumber(21);
        return  new ResponseEntity<>(ticketStatus, HttpStatus.CREATED);
    }

}
```




```
{
    "passengerId": 7402,
    "firstName": "Mohacel",
    "lastName": "Hosen",
    "phoneNumber": "01946363345",
    "journeyDate": "2023-05-10",
    "journeyFrom": "Dhaka",
    "journeyTo": "Netrokuna"
}
```

```
#################
HTTP PUT Request
#################

-> PUT request is used to update an existing record / resource at server

-> PUT Request can take data in URL and in Request Body

-> To bind our method to PUT request we will use @PutMapping

    @PutMapping("/ticket")
    public ResponseEntity<String> updateTicket(@RequestBody PassengerInfo request){
        System.out.println(request);
        //logic to update ticket
        return new ResponseEntity<>("Ticket Updated", HttpStatus.OK);
    }
```

```
@PutMapping("/ticket")
public ResponseEntity<String> updateTicket(@RequestBody PassengerInfo
passengerInfo){
    System.out.println(passengerInfo);
    // logic to update the ticket
    System.out.println("Passenger Data Updated Successfully...☑");

    return  new ResponseEntity<>("Updated Successfully...☑",HttpStatus.OK);
}
```

```
####################
HTTP DELETE Request
####################

-> DELETE request is used to delete an existing record / resource at server

-> DELETE Request can take data in URL and in Request Body

-> To bind our method to DELETE request we will use @DeleteMapping

    @DeleteMapping("/ticket/{ticketId}")
    public ResponseEntity<String> deleteTicket(@PathVariable("ticketId") Integer ticketId){
        //logic to delete the ticket
        return new ResponseEntity<>("Ticket Deleted", HttpStatus.OK);
    }
```

```
@DeleteMapping("/ticket/{ticketId}")
public ResponseEntity<String> deleteTicket(@PathVariable("ticketId") Integer
ticketId){
    System.out.println(ticketId);
    // logic to Delete the ticket
    System.out.println("Passenger Data Deleted Successfully...❌");

    return  new ResponseEntity<>("Deleted Successfully...❌",HttpStatus.OK);
}
```

springdoc-openapi

Add this dependency

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>

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

Create Config class

```
import io.swagger.v3.oas.annotations.OpenAPIDefinition;
import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Info;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@OpenAPIDefinition
public class SpringDocConfig {

    @Bean
    public OpenAPI baseOpenAPI(){
        return new OpenAPI().info(
                new Info().title("MCU CLUB DOC")
                        .version("1.0")
                        .description("MCU CLUB")
        );
    }
}
```

Use this link for UI

localhost:8080/swagger-ui/index.html

Complete REST API for USER

Step1: create a project with below dependency

1) spring-boot-starter-web
2) spring-boot-starter-data-jpa
3) mysql-connector-j
4) Lombok
5) spring-boot-devtools
6) springdoc-openapi-starter-webmvc-ui
7) spring-boot-starter-validation

Step2: create Entity & DTO class

```
package com.mohacel.education.entity;

import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "USER_TAB")
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "USER_ID_COL")
    private Integer id;
    @Column(name = "USER_NAME_COL")
    private String name;
    @Column(name = "USER_EMAIL_COL")
    private String email;
    @Column(name = "USER_GENDER_COL")
    private String gender;
    @Column(name = "USER_AGE_COL")
    private Integer age;
}
```

```
package com.mohacel.education.dto;

import lombok.Data;

@Data
public class UserDto {
    private Integer id;
    private String name;
```

```
    private String email;
    private String gender;
    private Integer age;
}
```

Step3: create a repository Interface that extends entity class

```
package com.mohacel.education.repository;

import com.mohacel.education.entity.UserEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<UserEntity, Integer> {
}
```

Step4: Define service method as abstract method inside interface

```
package com.mohacel.education.service;

import com.mohacel.education.dto.UserDto;

import java.util.List;

public interface IUserService {
    public UserDto saveUser(UserDto user);
    public UserDto updateUser(UserDto userDto, Integer id);
    public String deleteUser(Integer id);
    public List<UserDto> getAll();
    public UserDto getSingleUser(Integer id);

}
```

Step5: Provide Implementation of this abstract method

```
package com.mohacel.education.service;

import com.mohacel.education.dto.UserDto;
import com.mohacel.education.entity.UserEntity;
import com.mohacel.education.exception.UserResourcesNotFoundException;
import com.mohacel.education.repository.UserRepository;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;
@Service
public class UserServiceImpl implements IUserService{

    private UserRepository repository;
    public UserServiceImpl(UserRepository repository){
        this.repository =repository;
    }
    @Override
    public UserDto saveUser(UserDto user) {
        UserEntity userEntity = this.dto2user(user);
        UserEntity savedUser = this.repository.save(userEntity);
```

```
        return this.user2dto(savedUser);
    }

    @Override
    public UserDto updateUser(UserDto userDto, Integer id) {
        UserEntity user = repository.findById(id)
                .orElseThrow(()-> new UserResourcesNotFoundException("User",
"Id", id));

        user.setName(userDto.getName());
        user.setAge(userDto.getAge());
        user.setGender(userDto.getGender());
        user.setEmail(userDto.getEmail());

        UserEntity updatedUser = this.repository.save(user);
        UserDto dto = this.user2dto(updatedUser);
        return dto;
    }

    @Override
    public String deleteUser(Integer id) {
        UserEntity user = this.repository.findById(id).orElseThrow(() -> new
UserResourcesNotFoundException("User", "Id", id));
        this.repository.delete(user);
        return "User Deleted Successfully..";
    }

    @Override
    public List<UserDto> getAll() {
        List<UserEntity> entityList = this.repository.findAll();
        List<UserDto> userDtoList = entityList.stream().map(user ->
this.user2dto(user)).collect(Collectors.toList());
        return userDtoList;
    }

    @Override
    public UserDto getSingleUser(Integer id) {
        UserEntity user = repository.findById(id)
                .orElseThrow(()-> new UserResourcesNotFoundException("User",
"Id", id));

        return this.user2dto(user);
    }



    private UserEntity dto2user(UserDto userDto){
        UserEntity user = new UserEntity();
        user.setId(userDto.getId());
        user.setName(userDto.getName());
        user.setAge(userDto.getAge());
        user.setEmail(userDto.getEmail());
        user.setGender(userDto.getGender());
        return user;
    }

    private UserDto user2dto (UserEntity user){
```

```
        UserDto dto = new UserDto();
        dto.setId(user.getId());
        dto.setName(user.getName());
        dto.setAge(user.getAge());
        dto.setEmail(user.getEmail());
        dto.setGender(user.getGender());
        return dto;
    }
}
```

Step6: create Exception Class

```
package com.mohacel.education.exception;

public class UserResourcesNotFoundException extends RuntimeException{
   String resourceName;
   String fieldName;
   long fieldValue;

   public UserResourcesNotFoundException(String resourceName, String
fieldName, long fieldValue){
       super(String.format("%s not found with %s : %s",
resourceName,fieldName,fieldValue));
       this.resourceName=resourceName;
       this.fieldName=fieldName;
       this.fieldValue=fieldValue;

   }
}
```

```
package com.mohacel.education.response;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class ApiResponse {
    private String msg;
    private Boolean status;
}
```

```
package com.mohacel.education.exception;

import com.mohacel.education.response.ApiResponse;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserResourcesNotFoundException.class)
    public ResponseEntity<ApiResponse>
resourceNotFoundException(UserResourcesNotFoundException ex){
        String msg = ex.getMessage();
        ApiResponse response =  new ApiResponse(msg,false);
        return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
    }
}
```

Step7: create Rest Controller for this user

```
package com.mohacel.education.controller;

import com.mohacel.education.dto.UserDto;
import com.mohacel.education.service.IUserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private IUserService service;

    //POST--create user
    @PostMapping("/user")
    public ResponseEntity<UserDto> createUser(@RequestBody UserDto userDto){
        UserDto saveUser = service.saveUser(userDto);
        return  new ResponseEntity<>(saveUser, HttpStatus.CREATED);
    }


    //GET--get user
    @GetMapping("/user/{id}")
    public  ResponseEntity<UserDto> getById(@PathVariable Integer id){
        UserDto singleUser = service.getSingleUser(id);
        return new ResponseEntity<>(singleUser, HttpStatus.OK);
    }

    @GetMapping("/all")
```

```
    public  ResponseEntity<List<UserDto>> getAllUsers(){
        List<UserDto> allUsers = service.getAll();
        return new ResponseEntity<>(allUsers, HttpStatus.OK);
    }


    //PUT--update user
    @PutMapping("/user/{id}")
    public  ResponseEntity<UserDto> updateUser(@RequestBody UserDto userDto,
@PathVariable Integer id ){
        UserDto updateUser = service.updateUser(userDto, id);
        return  new ResponseEntity<>(updateUser, HttpStatus.OK);
    }

    //DELETE--delete user
    @DeleteMapping("/user/{id}")
    public  ResponseEntity<String> deleteUser(@PathVariable Integer id){
        String msg = service.deleteUser(id);
        return  new ResponseEntity<>(msg, HttpStatus.OK);
    }
}
```

Embedded Database (H2)

#####################
Embedded Database (H2)
#####################

-> Embedded Databases are temporary databases / in-memory databases

-> Embedded Databases are used for POC development (Proof of concept)

-> We no need to download and install embedded databases

-> Embedded Databases will come along with our application by adding one dependency

-> When we start application then embedded db will start and when we stop the application then embedded db will be stopped

Note: Data is not permenent in the embedded db (when we stop the application we will loose the data)

Note: In Memory DBs are not used for realtime project development in the company

-> We can use H2 DB as an in-memory db for practise purpose

H2 Database properties

```
spring.datasource.driver-class-name  = org.h2.Driver
spring.datasource.url  =  jdbc:h2:mem:studentDB
spring.datasource.password =  admin123
spring.datasource.username =  root

spring.jpa.show-sql  =  true
spring.jpa.hibernate.ddl-auto = update
```

Dependencies

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

Student Entity

```
package com.mohacel.student.entity;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "STUDENT_TABLE")
public class StudentEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_COL")
    private Integer id;
    @Column(name = "STD_ID_COL")
    private Integer stdId;
    @Column(name = "STD_Name_COL")
    private String name;
    @Column(name = "STD_GENDER_COL")
    private String gender;
    @Column(name = "STD_GMAIL_COL")
```

```
    private String email;
    @Column(name = "STD_DEPARTMENT_COL")
    private String department;
}
```

Student Repository

```
package com.mohacel.student.repository;

import com.mohacel.student.entity.StudentEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<StudentEntity,
Integer> {

}
```

IStudent Service

```
package com.mohacel.student.service;

import com.mohacel.student.entity.StudentEntity;

import java.util.List;

public interface IStudentService {
    public StudentEntity save(StudentEntity student);
    public StudentEntity update(StudentEntity student);
    public List<StudentEntity> getAll();
    public StudentEntity getSingleStudent(Integer id);
    public String deleteById(Integer id);
}
```

Student Service Impl

```
package com.mohacel.student.service.impl;

import com.mohacel.student.CustomException.IdNotFoundException;
import com.mohacel.student.entity.StudentEntity;
import com.mohacel.student.repository.StudentRepository;
import com.mohacel.student.service.IStudentService;
import org.springframework.stereotype.Service;

import java.util.List;
@Service
public class StudentServiceImpl implements IStudentService {

    private StudentRepository repository;


    public StudentServiceImpl(StudentRepository repository) {
        this.repository = repository;
        repository.save(new StudentEntity(1,1002,"Mohacel", "Male",
"mohacel.hosen06@gmail.com","CSE"));
```

```
    }

    @Override
    public StudentEntity save(StudentEntity student) {
        return repository.save(student);
    }

    @Override
    public StudentEntity update(StudentEntity student) {
        return repository.save(student);
    }

    @Override
    public List<StudentEntity> getAll() {
        return repository.findAll();
    }

    @Override
    public StudentEntity getSingleStudent(Integer id) {
        return repository.findById(id).orElseThrow(()-> new
IdNotFoundException("Id Not Found 😵"));
    }

    @Override
    public String deleteById(Integer id) {
        StudentEntity student = repository.findById(id).orElseThrow(() -> new
IdNotFoundException("Id Not Found 😵"));
        repository.delete(student);
        return "Deleted Successfully...";
    }
}
```

Custom Exception

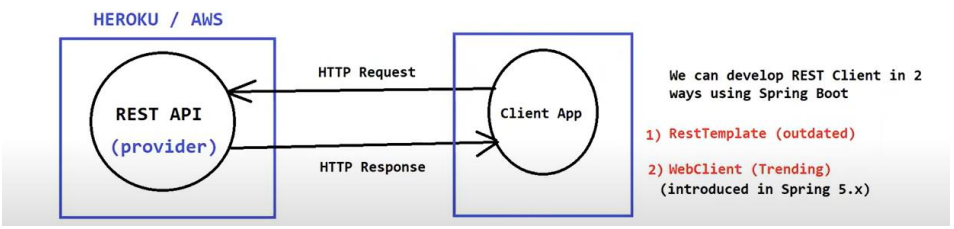
```
package com.mohacel.student.CustomException;

public class IdNotFoundException extends RuntimeException{
    public IdNotFoundException(){super();}
    public IdNotFoundException(String message){super(message);}
}
```

Student REST Controller

```
package com.mohacel.student.controller;

import com.mohacel.student.entity.StudentEntity;
import com.mohacel.student.service.impl.StudentServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
```

```java
@RestController
public class StudentController {
    @Autowired
    private StudentServiceImpl service;

    @GetMapping("/")
    public ResponseEntity<String> welcome(){
        String msg = "Welcome the world of REST API made by @MohacelHosen";
        return new ResponseEntity<>(msg,HttpStatus.OK);
    }

    @PostMapping("/student")
    public ResponseEntity<StudentEntity> saveStudentInfo(@RequestBody
StudentEntity student){
        StudentEntity studentEntity = service.save(student);
        return  new ResponseEntity<>(studentEntity, HttpStatus.CREATED);
    }

    @PutMapping("/student")
    public ResponseEntity<StudentEntity> updateStudentInfo(@RequestBody
StudentEntity student){
        StudentEntity updatedStudent = service.update(student);
        return  new ResponseEntity<>(updatedStudent, HttpStatus.OK);
    }

    @GetMapping("/student/{id}")
    public ResponseEntity<StudentEntity>
singleStudentInfo(@PathVariable("id") Integer id){
        StudentEntity singleStudent = service.getSingleStudent(id);
        return new ResponseEntity<>(singleStudent, HttpStatus.OK);
    }

    @GetMapping("/student")
    public ResponseEntity<List<StudentEntity>> AllStudentInfo(){
        List<StudentEntity> allStudent = service.getAll();
        return  new ResponseEntity<>(allStudent, HttpStatus.OK);
    }

    @DeleteMapping("student/{id}")
    public ResponseEntity<String> deleteStudentInfo(@PathVariable("id")
Integer id){
        String msg = service.deleteById(id);
        return  new ResponseEntity<>(msg, HttpStatus.OK);
    }
}
```

REST Client



################
REST Client
#################

-> The application which is accessing REST API is called as REST Client

-> Every Programming language having support to develop REST Client

-> Using Spring Boot also we can develop REST Client

1) RestTemplate

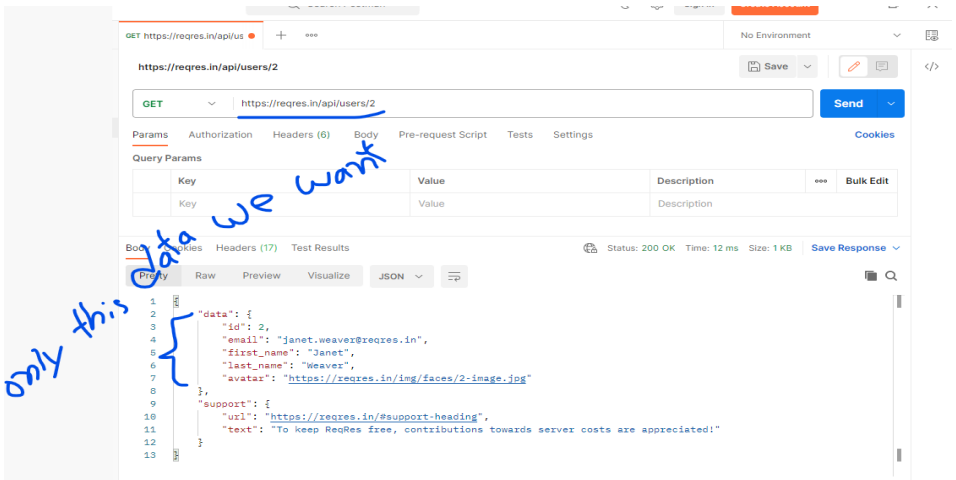
2) WebClient (introduced in spring 5.x version)

-> RestTemplate is a predefined class, which is part of 'spring-boot-starter-web' dependency. It supports only synchronus communication.

-> WebClient is a interface which is part of 'spring-boot-starter-webflux' dependency. It supports both synchronus & asynchronus communication.

Note: To develop REST CLient we need to know REST API details

1) API URL
2) API Request Method
2) API request data structure
3) API reponse data structure



Step1: Create a user model class that which property you want to receive

```
package com.mohacel.template.restClient;

import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.Data;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Data
@JsonIgnoreProperties(ignoreUnknown = true)
public class User {
    private Integer id;
    private String email;

    @JsonProperty("first_name")
    private String firstName;

    @JsonProperty("last_name")
    private String lastName;
}
```

Step2: inside a main class create **RestTemplate** object.

```
SpringApplication.run(RestTemplateApplication.class, args);
String apiUrl = "https://reqres.in/api/users/2";
RestTemplate template = new RestTemplate();
```

Step3: Create **HttpHeaders** object and set the accept type json

```
HttpHeaders headers = new HttpHeaders();
headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));
```

Step4: create **HttpEntity** object and pass the **HttpHeaders** object

```
HttpEntity<String> entity = new HttpEntity<>(headers);
ResponseEntity<String> response = template.exchange(apiUrl, HttpMethod.GET,
entity, String.class);
```

Step5: Create **ObjectMapper** object, it will help to convert json data to java object.

```
ObjectMapper mapper = new ObjectMapper();
```

Final step: Create a **JsonNode** object as root of the json and pass the response body of the **ResponseEntity<String>** and then use **get** method to get specific json data.

```
package com.mohacel.template;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.mohacel.template.restClient.User;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.*;
import org.springframework.web.client.RestTemplate;

import java.io.IOException;
import java.util.Collections;

@SpringBootApplication
public class RestTemplateApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestTemplateApplication.class, args);
        String apiUrl = "https://reqres.in/api/users/2";
        RestTemplate template = new RestTemplate();

        HttpHeaders headers = new HttpHeaders();

headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));

        HttpEntity<String> entity = new HttpEntity<>(headers);
        ResponseEntity<String> response = template.exchange(apiUrl,
HttpMethod.GET, entity, String.class);

        if (response.getStatusCode() == HttpStatus.OK) {
```

```
            ObjectMapper mapper = new ObjectMapper();
            try {
                JsonNode root = mapper.readTree(response.getBody());
                JsonNode dataNode = root.get("data");

                User user = mapper.treeToValue(dataNode, User.class);
                System.out.println(user);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Explanation:

1. The `main()` method serves as the entry point of the application.

2. `SpringApplication.run(RestTemplateApplication.class, args)` starts the Spring Boot application.

3. `String apiUrl = "https://reqres.in/api/users/2"` defines the URL of the API endpoint that you want to retrieve user data from.

4. `RestTemplate template = new RestTemplate()` creates an instance of `RestTemplate`, which is a class provided by Spring Framework for making HTTP requests.

5. `HttpHeaders headers = new HttpHeaders()` creates an instance of `HttpHeaders` to specify the headers for the HTTP request.

6. `headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON))` sets the `Accept` header to indicate that you expect the response to be in JSON format.

7. `HttpEntity<String> entity = new HttpEntity<>(headers)` creates an instance of `HttpEntity` to represent the entire HTTP request, including headers and body (in this case, there is no request body, so we pass an empty `String`).

8. `ResponseEntity<String> response = template.exchange(apiUrl, HttpMethod.GET, entity, String.class)` performs the HTTP GET request to the specified API URL with the given headers and retrieves the response as a `ResponseEntity<String>`.

9. The `if (response.getStatusCode() == HttpStatus.OK)` condition checks if the response status is `200 OK`.

10. `ObjectMapper mapper = new ObjectMapper()` creates an instance of `ObjectMapper`, which is responsible for mapping JSON data to Java objects.

11. `JsonNode root = mapper.readTree(response.getBody())` reads the JSON response body and creates a `JsonNode` representing the root of the JSON tree.

12. `JsonNode dataNode = root.get("data")` retrieves the "data" node from the JSON response.

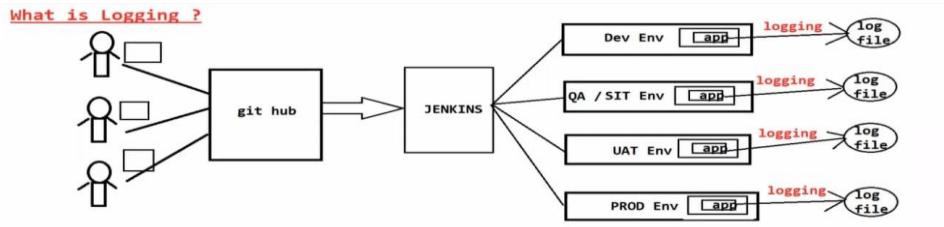
13. `User user = mapper.treeToValue(dataNode, User.class)` converts the `dataNode` JSON subtree to a `User` object using the `treeToValue()` method of `ObjectMapper`.

14. `System.out.println(user)` prints the `User` object to the console.
15. `catch (IOException e)` handles any `IOException` that might occur during JSON processing and prints the stack trace.

This code makes an HTTP GET request to the specified API endpoint, retrieves the JSON response, and maps it to a `User` object using Jackson's `ObjectMapper`. Finally, it prints the `User` object to the console.

Note: Make sure you have the necessary dependencies, such as `spring-boot-starter-web` and `jackson-databind`, in your project's build configuration.

Logging for Project



```
###########
Logging
###########

-> Logging is a process of storing application execution details

-> Log messages will help us to understand the problems occuring in the application execution

-> If any problem occurs in  project execution we need to check log file of that project to find out the root cause

-> In Realtime, for every project Logging will be implemented

                                logging                          check logs
          application ---------------------> Log file <------------------------------developers
```

Log Architecture

```
1) Logger   : It is a API  which provides methods to perform logging

          Ex : trace ( ),  debug ( ) , info ( ), warn ( ), error ( ) , fatal ( ) etc....

2) Layout : It represents log msg format (what info shud present in log msg)

          Ex:  DATE TIME  ClassName  Method Name : Log Msg

3) Appender   : It is used to write the log msg to destination

          Ex: Console Appender to write log msg to console

              FileAppender to write log msg to log file

              RollingFileAppender to write log msgs to log file based on Rolling Policies

                        - TimeBasedRolling    (Based on given time new log file will be created)
                        - SizeBasedRolling  (Based on given size new log file will be created)
```

Step1: create a logger object inside rest controller class

```
Logger logger =
LoggerFactory.getLogger(UserRestController.class);
```

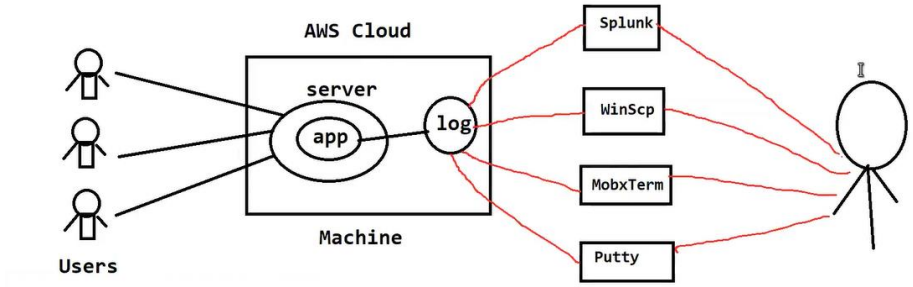
```
package com.mohacel.club.controller;

import com.mohacel.club.entity.UserRegistrationEntity;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;


@RestController
public class UserRestController {
    Logger logger = LoggerFactory.getLogger(UserRestController.class);
    @GetMapping("/")
    public ResponseEntity<UserRegistrationEntity> getMember(){
        logger.info("getMember() method execution start....");
```

```
        UserRegistrationEntity user1 = new
UserRegistrationEntity(1002,"Mohacel","Male","mohacel.hosen06@gmail.com");
        logger.info("getMember() method execution end");

        return  new ResponseEntity<>(user1, HttpStatus.OK);
    }
}
```

Step2: Set the log properties into application.properties

```
logging.file.name=mcu.log
logging.logback.rollingpolicy.file-name-
pattern=mcu-%d{yyyy-MM-dd}.%i.log
logging.logback.rollingpolicy.max-file-size=1MB
logging.logback.rollingpolicy.max-history=50
logging.logback.rollingpolicy.total-size-
cap=10MB
```

To see the log file from a remote server



Connect Linux server using Splunk/ WinScp/ MobxTerm/ Putty then check your log file name.

- ⇨ *ls -a*
- ⇨ *tail -n 100 mcuClub.log*

```
############
Log Levels
###########

-> Log level will represent priotiry of log msg

        TRACE   >   DEBUG  >   INFO    >   WARN   >   ERROR  >   FATAL

-> Log messages we will generate using Log Levels

-> Based On Log Level Configured, the log msgs will be printed

-> From the configured level to higher level logs will be printed

Note: In Spring Boot the default log level is 'INFO'
```

Exception Handling

```
############################
Exception Handling In REST API
############################

-> Exception is an unexpected and unwanted situation occuring in the application

-> When exception occured our program will terminate abnormally

-> To achieve graceful termination of the program we need to handle the exception

-> In Java we have below keywords to handle the exceptions

1) try   :  It is used to keep risky code

2) catch  : Catch block is used to handle the exception

3) throw  : It is used to re-throw the exception

4) throws : It is used to ignore the exception

5) finally :  It is used to execute clean up logic (closing files, closing connection, release resources....)
```

Note: When we get exception in REST API we should convey that exception information to client / client application in json format

Ex:

```
{
    msg : "Exception Reason"
    code : "SBI0004"
}
```

Note: In project for every exception we will use on CODE i.e exception code

```java
package com.mohacel.club.controller;

import com.mohacel.club.exception.ExceptionInfo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class WelcomeController {
    Logger logger = LoggerFactory.getLogger(WelcomeController.class);

    @GetMapping("/welcome")
    public ResponseEntity<String> greet(){
        String wish = "Welcome the world of Spring Boot";
        try {
            int result = 10/0;
        }catch (Exception e){
            logger.info("Exception Occurred ::"+e.getMessage());
            throw  new ArithmeticException(e.getMessage());
        }
        return  new ResponseEntity<>(wish, HttpStatus.OK);
    }

    @ExceptionHandler(value = ArithmeticException.class)
    public ResponseEntity<ExceptionInfo> handleAE(ArithmeticException ae){
        ExceptionInfo exceptionInfo = new ExceptionInfo();
        exceptionInfo.setErrorCode(ae.getMessage());
        exceptionInfo.setErrorCode("AEB101");
        return  new ResponseEntity<>(exceptionInfo,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Download Report Data as PDF and Excel

Add these two dependencies (poi-ooxml and openpdf)

```
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.2.3</version>
</dependency>
<dependency>
    <groupId>com.github.librepdf</groupId>
    <artifactId>openpdf</artifactId>
    <version>1.3.30</version>
</dependency>
```

Add Entity Class from which table you want to extract the data

```
package com.mohacel.covid.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data;

@Data
@Entity(name = "COVID_REPORT_TABLE")
public class CovidReport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer covidStatisticId;
    private String millionPerPopulationCases;
    private Integer activeCases;
    private Integer criticalCases;
    private String newCases;
    private Integer recoveredCases;
    private Integer totalCases;
    private String continent;
    private String country;
    private String collectedDay;
    private String millionPerPopulationDeaths;
    private String newDeaths;
    private Integer totalDeaths;
    private Integer population;
    private String millionPerPopulationTests;
    private Integer totalTests;
}
```

Create Repository Interface For this entity class

```
package com.mohacel.covid.repository;

import com.mohacel.covid.entity.CovidReport;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;
```

```
public interface CovidReportRepo extends JpaRepository<CovidReport, Integer>
{

    List<String> getDistinctByCountry();

    List<String> getDistinctByContinent();
}
```

Create IService Class

```
package com.mohacel.covid.service;

import com.mohacel.covid.dto.SearchResponse;
import com.mohacel.covid.request.SearchRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.util.List;

public interface ICovidReportService {
    public List<String> getUniqueCountry();
    public List<String> getUniqueContinent();
    public List<SearchResponse>  search(SearchRequest request);
    public void generateExcel(HttpServletResponse response) throws
IOException;
    public void generatePdf(HttpServletResponse response) throws IOException;
}
```

Implement the IService Class's methods

```
package com.mohacel.covid.service.impl;

import com.lowagie.text.*;
import com.lowagie.text.Font;
import com.lowagie.text.pdf.PdfPCell;
import com.lowagie.text.pdf.PdfPTable;
import com.lowagie.text.pdf.PdfTable;
import com.lowagie.text.pdf.PdfWriter;
import com.mohacel.covid.dto.SearchResponse;
import com.mohacel.covid.entity.CovidReport;
import com.mohacel.covid.repository.CovidReportRepo;
import com.mohacel.covid.request.SearchRequest;
import com.mohacel.covid.service.ICovidReportService;
import jakarta.servlet.ServletOutputStream;
import jakarta.servlet.http.HttpServletResponse;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Example;
import org.springframework.data.domain.Sort;
```

```java
import org.springframework.stereotype.Service;

import java.awt.*;
import java.io.IOException;
import java.lang.annotation.Documented;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.stream.Collectors;

@Service
public class CovidReportServiceImplementation implements ICovidReportService
{
    @Autowired
    private CovidReportRepo covidReportRepo;

    @Override
    public List<String> getUniqueCountry() {
        return covidReportRepo.getDistinctByCountry();
    }

    @Override
    public List<String> getUniqueContinent() {
        return covidReportRepo.getDistinctByContinent();
    }

    @Override
    public List<SearchResponse> search(SearchRequest request) {
        SearchRequest newSearchRequest = new SearchRequest();

        String continent = request.getContinent();
        if (continent != null && !continent.equals("")) {
            newSearchRequest.setContinent(continent);
        }

        String country = request.getCountry();
        if (country != null && !country.equals("")) {
            newSearchRequest.setCountry(country);
        }

        String collectedDay = request.getCollectedDay();
        if (collectedDay != null && !collectedDay.equals("")) {
            newSearchRequest.setCollectedDay(collectedDay);
        }

        Example<SearchRequest> example = Example.of(newSearchRequest);

        //query by example is used for dynamic queries where example act as
where clues
        List<CovidReport> reportList = covidReportRepo.findAll((Sort)
example);

        return reportList.stream()
                .map(x -> new SearchResponse(
                        x.getCollectedDay(),
                        x.getContinent(),
                        x.getCountry(),
                        x.getPopulation(),
```

```
                        x.getTotalCases(),
                        x.getTotalTests(),
                        x.getTotalDeaths()))
                .collect(Collectors.toList());
    }


    // Generate Excel with appropriate data
    @Override
    public void generateExcel(HttpServletResponse response) throws
IOException {

        List<CovidReport> covidReportList = covidReportRepo.findAll();
        HSSFWorkbook workbook = new HSSFWorkbook();
        HSSFSheet sheet = workbook.createSheet();
        HSSFRow headerRow = sheet.createRow(0);

        headerRow.createCell(0).setCellValue("collectedDay");
        headerRow.createCell(1).setCellValue("continent");
        headerRow.createCell(2).setCellValue("country");
        headerRow.createCell(3).setCellValue("population");
        headerRow.createCell(4).setCellValue("totalCases");
        headerRow.createCell(5).setCellValue("totalTests");
        headerRow.createCell(6).setCellValue("totalDeaths");

        AtomicInteger rowNumber = new AtomicInteger(1);

        covidReportList.forEach(entity -> {
            HSSFRow dataRow = sheet.createRow(rowNumber.getAndIncrement());
            dataRow.createCell(0).setCellValue(entity.getCollectedDay());
            dataRow.createCell(1).setCellValue(entity.getContinent());
            dataRow.createCell(2).setCellValue(entity.getCountry());
            dataRow.createCell(3).setCellValue(entity.getPopulation());
            dataRow.createCell(4).setCellValue(entity.getTotalCases());
            dataRow.createCell(5).setCellValue(entity.getTotalTests());
            dataRow.createCell(6).setCellValue(entity.getTotalDeaths());

        });

        ServletOutputStream outputStream = response.getOutputStream();
        workbook.write(outputStream);
        workbook.close();
        outputStream.close();

    }


    // Generate PDF with appropriate data
    @Override
    public void generatePdf(HttpServletResponse response) throws IOException
{
        List<CovidReport> covidReportList = covidReportRepo.findAll();

        Document document = new Document(PageSize.A4);
        PdfWriter.getInstance(document, response.getOutputStream());
        document.open();
```

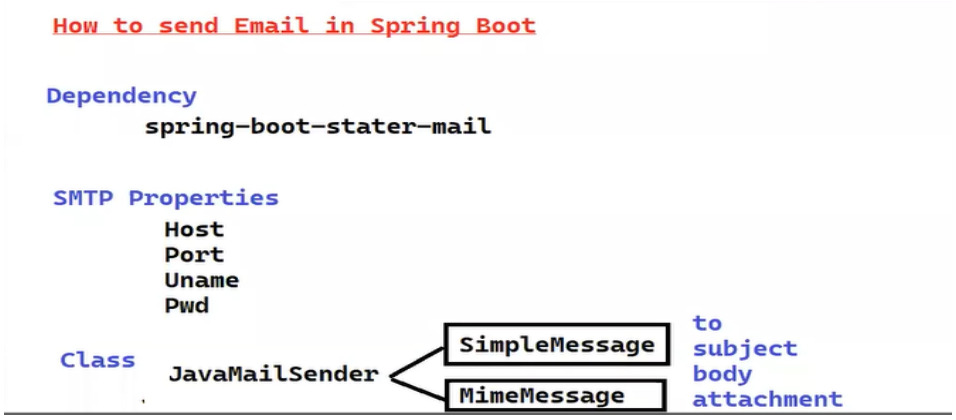
```java
        Font font = FontFactory.getFont((FontFactory.HELVETICA_BOLD));
        font.setSize(18);
        font.setColor(Color.BLACK);

        Paragraph p = new Paragraph("Covid Statistic Report", font);
        p.setAlignment(Paragraph.ALIGN_CENTER);
        document.add(p);

        PdfPTable table = new PdfPTable(7);
        table.setWidthPercentage(100f);
        table.setWidths(new float[]{3f, 2f, 3.0f, 3.0f, 3.0f, 3.0f, 3.0f});
        table.setSpacingBefore(10);

        PdfPCell cell = new PdfPCell();
        cell.setBackgroundColor(Color.LIGHT_GRAY);
        cell.setPadding(5);

        font = FontFactory.getFont(FontFactory.HELVETICA_BOLD);
        font.setColor(Color.BLACK);

        cell.setPhrase(new Phrase("Collected Date", font));
        table.addCell(cell);

        cell.setPhrase(new Phrase("Continent", font));
        table.addCell(cell);

        cell.setPhrase(new Phrase("Country", font));
        table.addCell(cell);

        cell.setPhrase(new Phrase("Population", font));
        table.addCell(cell);

        cell.setPhrase(new Phrase("Total Cases", font));
        table.addCell(cell);

        cell.setPhrase(new Phrase("Total Tests", font));
        table.addCell(cell);

        cell.setPhrase(new Phrase("Total Deaths", font));
        table.addCell(cell);

        for (CovidReport entity : covidReportList) {
            table.addCell(entity.getCollectedDay());
            table.addCell(entity.getContinent());
            table.addCell(entity.getCountry());
            table.addCell(String.valueOf(entity.getPopulation()));
            table.addCell(String.valueOf(entity.getTotalCases()));
            table.addCell(String.valueOf(entity.getTotalTests()));
            table.addCell(String.valueOf(entity.getTotalDeaths()));
        }
        document.add(table);
        document.close();
    }
}
```

Create Rest Controller

```
package com.mohacel.covid.controller;

import com.mohacel.covid.dto.SearchResponse;
import com.mohacel.covid.request.SearchRequest;
import com.mohacel.covid.service.impl.CovidReportServiceImplementation;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.io.IOException;
import java.util.List;

@RestController
public class CovidReportController {
    @Autowired
    CovidReportServiceImplementation reportService;

    @GetMapping("/continent")
    public ResponseEntity<List<String>> getContinentName(){
        List<String> uniqueContinent = reportService.getUniqueContinent();
        return new ResponseEntity<>(uniqueContinent, HttpStatus.OK);
    }

    @GetMapping("/country")
    public ResponseEntity<List<String>> getCountryName(){
        List<String> uniqueCountry = reportService.getUniqueCountry();
        return new ResponseEntity<>(uniqueCountry, HttpStatus.OK);
    }

    @PostMapping("/search")
    public ResponseEntity<List<SearchResponse>> searchCovidData(@RequestBody
SearchRequest request){
        List<SearchResponse> searchResponseList =
reportService.search(request);
        return new ResponseEntity<>(searchResponseList, HttpStatus.OK);
    }

    @GetMapping("/excel")
    public void excelReport(HttpServletResponse response) throws IOException
{
        response.setContentType("application/octet-stream");

        String header_key = "Content-Disposition";
        String header_value = "attachment;filename=covidReport.xls";

        response.setHeader(header_key, header_value);
        reportService.generateExcel(response);
    }
```

```
    @GetMapping("/pdf")
    public void pdfReport(HttpServletResponse response) throws IOException {
        response.setContentType("application/pdf");

        String header_key = "Content-Disposition";
        String header_value = "attachment;filename=covidReport.pdf";
        response.setHeader(header_key, header_value);
        reportService.generatePdf(response);
    }
}
```

Email in Spring Boot



Step1: add the dependency

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

Step2: Create a class and autowired this interface "JavaMailSender"

Commented [MH1]: It's a predefine interface available "**org.springframework.mail.javamail.JavaMailSender**; " inside this package

```
@Autowired
private JavaMailSender mailSender;
```

Step3: create a method inside the class with "String to, String subject, String body" this parameter.

```
package com.mohacel.user.utils;

import jakarta.mail.internet.MimeMessage;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Component;

@Component
public class EmailUtils {
    @Autowired
    private JavaMailSender mailSender;

    public boolean sendEmail(String to, String subject, String body) {
        boolean isMailSent = false;

        try {
            MimeMessage mimeMessage = mailSender.createMimeMessage();
            MimeMessageHelper messageHelper = new
MimeMessageHelper(mimeMessage);
            messageHelper.setTo(to);
            messageHelper.setSubject(subject);
            messageHelper.setText(body, true);
            mailSender.send(mimeMessage);
            isMailSent = true;
        } catch (Exception e) {
            e.printStackTrace();
        }

        return isMailSent;
    }
}
```

MimeMessage is used for attachment and we can also send the link on MimeMessage even we can send HTML file which is not possible SimpleMessage

Step4: create a file inside project folder with proper response message

```
Hi {FULLNAME},<br/>
Your registration is almost completed. <br/>
Please find your temporary password below <br/>
Temporary Password: {TEMP-PWD} <br/>
<a href="{URL}"> Click Here to Active </a>

Thanks,
Admin Team.
```

Step5: Create a method to read this file

```
private String readEmailBody(String fullName, String tempPwd, String
fileName) throws IOException {
    String url = "";
    String mailBody = "";

    FileReader fileReader = new FileReader(fileName);
    BufferedReader bufferedReader = new BufferedReader(fileReader);

    StringBuilder stringBuilder = new StringBuilder();

    String readLine = bufferedReader.readLine();
    while (readLine != null) {
        stringBuilder.append(readLine);
        readLine = bufferedReader.readLine(); // Update the readLine value
    }
    bufferedReader.close();
    mailBody = stringBuilder.toString();
    mailBody = mailBody.replace("{FULLNAME}", fullName); // Assign the
updated value back to mailBody
    mailBody = mailBody.replace("{TEMP-PWD}", tempPwd); // Assign the updated
value back to mailBody
    mailBody = mailBody.replace("{URL}", url); // Assign the updated value
back to mailBody

    return mailBody;
}
```

Step6: call the method inside saveUser / forgetPassword

```
@Override
public boolean saveUserDetails(User user) throws IOException {

    UserDetailsEntity entity = new UserDetailsEntity();
    BeanUtils.copyProperties(user, entity);

    entity.setPassword(generateRandomString(6));
    entity.setAccountStatus(false);

    //TODO:: send email with temp password
    String subject = "Registration Message to Activate the Account";
    String fileName = "REG-EMAIL-BODY.txt";
    String body = readEmailBody(entity.getFullName(), entity.getPassword(),
fileName);

    emailUtils.sendEmail(user.getEmail(), subject, body);

    return userDetailsRepo.save(entity).getUserId() != null;
```

```
}
```

```
// to generate random password
private String generateRandomString(int length) {
    final String ALPHANUMERIC_CHARS =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()";

    StringBuilder sb = new StringBuilder(length);
    Random random = new Random();

    for (int i = 0; i < length; i++) {
        int randomIndex = random.nextInt(ALPHANUMERIC_CHARS.length());
        char randomChar = ALPHANUMERIC_CHARS.charAt(randomIndex);
        sb.append(randomChar);
    }
    return sb.toString();
}
```

User Rest Controller

```
package com.mohacel.user.controller;

import com.mohacel.user.dto.ActivateAccount;
import com.mohacel.user.dto.Login;
import com.mohacel.user.dto.User;
import com.mohacel.user.service.imple.UserServiceImplementation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.io.IOException;
import java.util.List;

@RestController
public class UserRestController {
    @Autowired
    private UserServiceImplementation service;


    //save the user info
    @PostMapping("/user")
    public ResponseEntity<String> userRegistration(@RequestBody User user)
throws IOException {

        boolean isSaved = service.saveUser(user);

        if (isSaved){
```

```
            return  new ResponseEntity<>("Registration Successfully",
HttpStatus.CREATED);
        }else{
            return new ResponseEntity<>("Registration Fail",
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }


    // active the account
    @PostMapping("/activate")
    public ResponseEntity<String>  activeAccount(@RequestBody ActivateAccount
account){
        boolean isActive = service.activateUserAcc(account);

        if (isActive){
            return  new ResponseEntity<>("Account Activated", HttpStatus.OK);
        }else{
            return  new ResponseEntity<>("Invalid Temporary Password",
HttpStatus.BAD_REQUEST);
        }
    }

    // retrieve all the user
    @GetMapping("/users")
    public ResponseEntity<List<User>> getAllUser(){
        List<User> userList = service.getAllUser();
        return  new ResponseEntity<>(userList, HttpStatus.OK);
    }

    @GetMapping("/user/{userId}")
    public ResponseEntity<User> getSingleUser(@PathVariable Integer userId){
        User user = service.getUserById(userId);
        return  new ResponseEntity<>(user, HttpStatus.OK);
    }

    // delete the user by userId
    @DeleteMapping("/user/{userId}")
    public ResponseEntity<String> deleteUserById(@PathVariable Integer
userId){
        boolean isDeleted = service.deleteUserById(userId);
        if (isDeleted){
            return new ResponseEntity<>("Deleted Successfully",
HttpStatus.OK);
        }else {
            return  new ResponseEntity<>("Failed to Delete",
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    // change Account status
    @GetMapping("/status/{userId}/{status}")
    public ResponseEntity<String> statusChange(Integer userId, boolean
status){
        boolean isStatusChange = service.changeAccountStatus(userId, status);

        if (isStatusChange){
```

```
            return  new ResponseEntity<>("Status changed", HttpStatus.OK);
        }else{
            return new ResponseEntity<>("Failed to change",
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }


    // for login
    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody Login login){
        String response = service.login(login);
        return new ResponseEntity<>(response, HttpStatus.OK);
    }

    //forget password

    @GetMapping("/forget/{email}")
    public ResponseEntity<String> forgetPassword(@PathVariable String email)
throws IOException {
        String response = service.forgetPassword(email);
        return  new ResponseEntity<>(response, HttpStatus.OK);
    }

}
```

Sonarqube

- ⇨ **Create Ec2 Instance with 4 GB RAM (t2.medium) /**
- ⇨ **Create Azure Virtual Machine x64 Linux**
- ⇨ **Connect with VM**

Command

- ⇨ *$sudo apt update*
- ⇨ *$sudo apt install unzip*
- ⇨ *$sudo su*

- ⇨ *$cd /opt*
- ⇨ *$sudo apt install openjdk-17-jdk*
- ⇨ *$wget https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-9.9.1.69595.zip*
- ⇨ *$unzip sonarqube-*.zip*
- ⇨ *$sudo mv sonarqube-*/ /opt/sonarqube*

Create New User

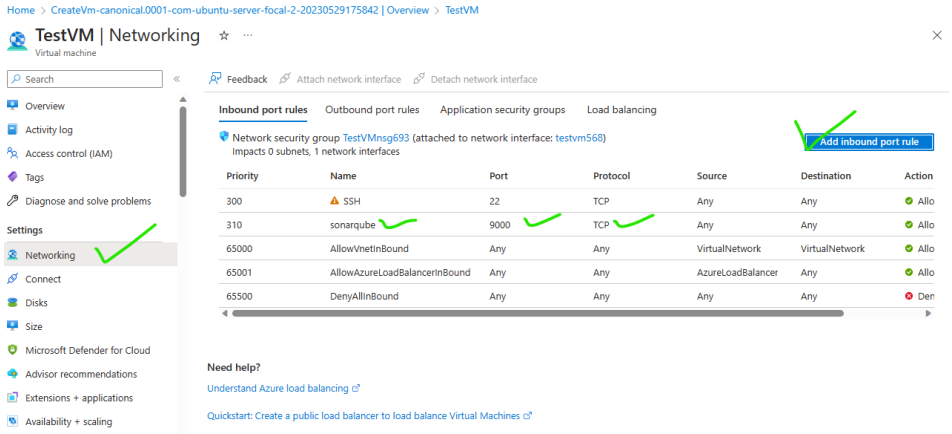
- ⇨ *$useradd <userName> (ex: useradd sonar)*
- ⇨ *$visudo (configure user without password)*
- ⇨ ***sonar ALL=(ALL) NOPASSWD: ALL***

```
## Next comes the main part: which users can run what software on
## which machines (the sudoers file can be shared between multiple
## systems).
## Syntax:
##
##      user    MACHINE=COMMANDS
##
## The COMMANDS section may have other options added to it.
##
## Allow root to run any commands anywhere
root    ALL=(ALL)       ALL
sonar ALL=(ALL) NOPASSWD: ALL
## Allows members of the 'sys' group to run networking, software,
## service management apps and more.
# %sys ALL = NETWORKING, SOFTWARE, SERVICES, STORAGE, DELEGATING, PROCESSES, LOCATE, DRIVERS

## Allows people in group wheel to run all commands
%wheel  ALL=(ALL)       ALL

## Same thing without a password
# %wheel        ALL=(ALL)       NOPASSWD: ALL

## Allows members of the users group to mount and unmount the
## cdrom as root
# %users  ALL=/sbin/mount /mnt/cdrom, /sbin/umount /mnt/cdrom

## Allows members of the users group to shutdown this system
# %users  localhost=/sbin/shutdown -h now
```

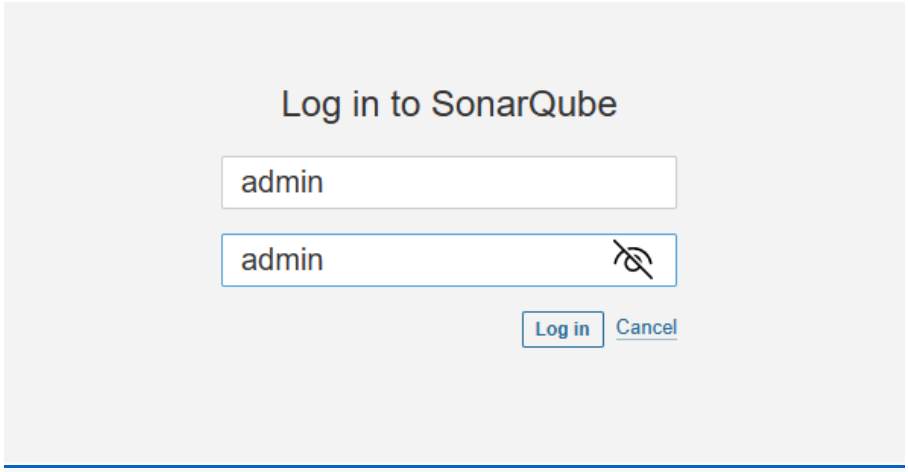
Change Ownership of this folder

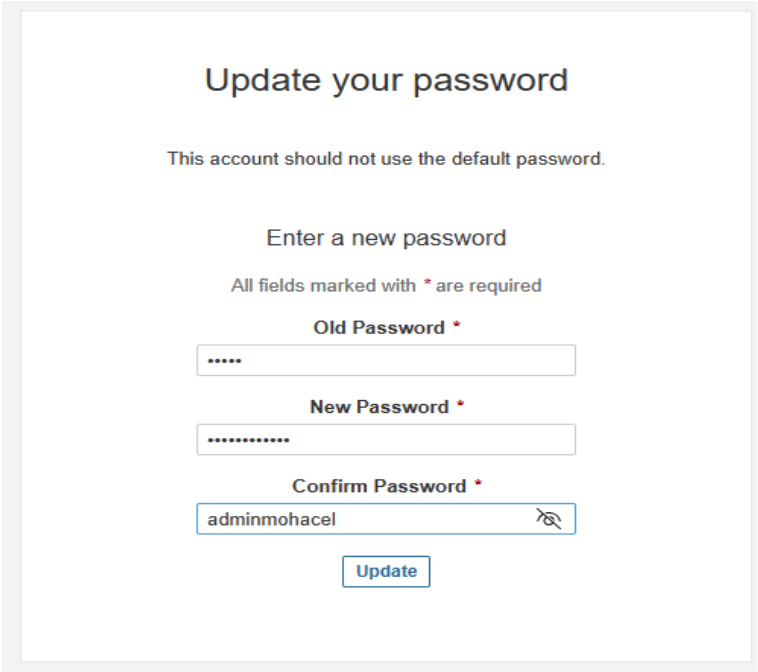
- ⇨ *$ cd /opt*
- ⇨ *$ chown -R sonar:sonar /opt/sonarqube/*
- ⇨ *$ chmod -R 775 /opt/sonarqube*
- ⇨ *$ su – sonar*
- ⇨ *$ cd sonarqube/bin/linux-x86-64/*
- ⇨ *$ sh sonar.sh start*
- ⇨ *$sh sonar.sh status*

```
sonar@TestVM:/opt/sonarqube/bin/linux-x86-64$ ls -l
total 8
-rwxrwxr-x 1 sonar sonar 7133 Apr 27 13:06 sonar.sh
sonar@TestVM:/opt/sonarqube/bin/linux-x86-64$ sh sonar.sh start
/usr/bin/java
Starting SonarQube...
Started SonarQube.
sonar@TestVM:/opt/sonarqube/bin/linux-x86-64$ ^C
sonar@TestVM:/opt/sonarqube/bin/linux-x86-64$ sh sonar.sh status
/usr/bin/java
SonarQube is running (18196).
sonar@TestVM:/opt/sonarqube/bin/linux-x86-64$
```



Access the sonarqube: **http://13.92.154.186:9000/**





Add this properties inside pom.xml

```
<properties>
    <sonar.host.url>http://13.92.154.186:9000/</sonar.host.url>
    <sonar.login>admin</sonar.login>
    <sonar.password>adminmohacel</sonar.password>
</properties>
```

Execute below maven on the terminal

- ⇨ *mvn clean package*
- ⇨ *mvn sonar:sonar*

Step to develop the full-stack Application using Angular

Step1: create spring boot application with below dependencies

i) spring-boot-starter-web
ii) spring-boot-devtools

step2: create @RestController method

```
package com.mohacel.fullstackapp.controller;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin
public class WelcomeController {
```

```
    @GetMapping("/welcome")
    public String welcome(){
        String msg = "Welcome the world of Full Stack Application";
        return msg;
    }
}
```

step3: Run the spring Boot Application

step4: Create Angular Application using VSCode

⇨ ng new app1

step5: import **HttpClientModule** in AppModule.ts file

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {HttpClientModule} from '@angular/common/http'

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

step6: Create function in App component class to handle template request.

```
import { Component, Inject } from '@angular/core';
import {HttpClient} from '@angular/common/http'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
```

```
})
export class AppComponent {
  title = 'First Full-Stack Application';
  message:string="I am respond from Angular";

  //depende
constructor(@Inject (HttpClient)  private http:HttpClient){}

  getMessage(){
      this.http.get('http://localhost:8080/welcome', {responseType:"text"})
      .subscribe(response=>{
        this.message=response;
      })
  }


  greetMessage(){
    this.http.get('http://localhost:8080/greet', {responseType:'text'})
    .subscribe(Response=>{
      this.message=Response;
    })
  }
}
```

Step7: Design presentation logic in template file.

```
<h1>{{title}}</h1>

<button (click)="getMessage()">Get The Message</button>
<button (click)="greetMessage()">Greet</button>
<br>
<br>
<h2>{{message}}</h2>
```