Spring Boot with Cloud & Microservices (19-SBMS)
++

=> What are the pre-requisites to learn this course ?

=> What is our SBMS course content ?

=> Who should learn this SBMS course ?

=> Why to learn this SBMS course ?

=> How job opportunites are available for Spring Boot & Microservices

=> SBMS Course Details

Pre-Requisites
++++++++++++++

1) Core Java

2) Adv Java (JDBC, Servlets & JSP)

3) SQL (CRUD)

SBMS Course Content
+++++++++++++++++++

1) Spring Core Module

2) Spring Boot

3) Spring Data JPA

4) Spring Web MVC

5) RESTFul Services (Spring REST)

6) Spring Cloud

7) Microservices

8) Spring Security

Who should learn this SBMS ?
++++++++++++++++++++++++++++

-> Every java developer should learn Spring Boot & Microservices
(both freshers & experienced)

-> To get job as a java developer we need to have very good knowledge in SBMS

-> To survive in IT as a java developer we need to have very good knowledge in SBMS

-> Now a days all java projects are using SBMS only

-> Every company asking for Spring Boot & Microservices resources only

====> For 3 years java developer with SBMS ===> 15+ lakhs package

Your Trainer Info
++++++++++++++++
Name : Mr. Ashok
Exp : 9+ Yrs Industry Experience
Role : Tech Lead
USA Based Banking Company (Product Based)

Course Info
+++++++++++
Name : Spring Boot with Cloud & Microservices
Course Code : 19-SBMS
Start Date : 01-Jun-2022
Class Time : 7 AM - 8:15 AM IST (Mon - Sat)
Duration : 3.5 Months
Course Fee : 7,000 INR (Live Classes + Daily ClassNotes)
For Backup Videos : 3,000 INR (6 months access)

Note: After completion of SBMS, you are equal to 4+ years exp Spring Boot developer.

++

Application Development
+++++++++++++++++++++++

=> Application development divied into 2 parts

1) Backend Development

2) Frontend Development

######## Fullstack Development = Backend development + Frontend development #########

=> Backend contains the business logic of our application

(webservice calls, validations, email logic, db communication)

=> Frontend contains user interface of the application

(presentation logic)

=> End users interact with frontend of our application. Frontend will interact with Bakend of our application. Backend will execute business logic and it will communicate with database also.

=> The developers who can do both backend development and front end development they are called as Fullstack Developers.

=> Fullstack developers are having lot of demand in the market.

=> Fullstack developers are hot cakes in the market

=> Companies are offering high packages for fullstack developers.

++
Monolith Vs Microservices
++

=> If we develop all the functionalities in single project then it is called as Monolith Architecture based application.

=> In Microservices architecture, functionality will be divided into several apis.

+++++++++++++++
Java Road Map
+++++++++++++

=> Core Java

=> Adv Java
```
            - JDBC
            - Servlets
            - JSP
```

=> SQL

=> Data Structures & Algorithms

=> Hibernate ORM Framework

=> Spring Framework

=> Spring Boot

=> RESTFul Services

=> Microservices

=> Security

=> Design Patterns

=> Spring Boot is just one approach to develop Spring Based applications with less configurations.

=> Spring Boot came into market with Auto Configuration concept

Note: Spring Boot will take care of configurations required for our application.

=> We can do rapid development using Spring Boot

UI Technologies
+++++++++++++++
HTML & CSS
Java Script
Boot Strap
Angular
React JS

Realtime Tools
++++++++++++++
Maven
Git Hub
JIRA

Spring Framework
----

1) What is Programming Language ?

2) What is Framework ?

-> Programming Language used by humans to communicate with Computers.
-> Programming Language contains set of instructions & syntaxes

Ex : C, C++, Java, C#, Python etc.....

-> 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 to focus only on business logic
(frameworks will provide common logics required for our application)

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

Struts : It is a web framework. Used to develop only 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)

Spring Advantages

-> 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

```
            POJO : Plain old java object
            POJI : Plain old java interface
```

-> Spring is not a single framework. It is collection of Modules

----
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 to deal with Cross Cutting logics in application.

Application = Business Logic + Cross Cutting Logic

Note: We can seperate 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.

-> Spring Framework released in 2004 (First Production Released)

-> The current version of Spring is 5.x version (Released in 2017)

Note: Reactive Programming support added in Spring Framwork 5.x version

Note: Spring Boot 1.x released in 2014

-> The current version of Spring Boot is 3.x version

Note: Spring Boot is an extension for Spring Framework.

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 Managaing dependencies among the classes

(Creating Objects & Injecting Objects)

-> 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) Inheritence
2) Composition

-> If we use Inheritence 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 practises.

-> "Spring Core" suggesting Developers 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 Behavioural Design Pattern

-> It enables our code to choose an alogrithm at run time

-> When we have multiple algorithms then we can use this pattern to choose one algorithm at run time

-> It will make our classes loosely coupled

Rules
+++++

1) Favour Compositon over inhertience

2) Code to interfaces instead of implementation classes

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

++
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 ::

```
                         BillCollector bc = new BillCollector();
                         bc.setPayment(new CreditCardPayment());
```

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

Ex::

```
                 BillCollector bc1 = new BillCollector(new
DebitcardPayment());
```

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

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

```
                 ex : obj.variable = value; //initialization
```

Note: If variable is declared as private then we can't access that variable outside of the class directley. To access private variables outside of the class we can use Reflection api.

```
package in.ashokit;

import java.lang.reflect.Field;

public class Test {

      public static void main(String[] args) throws Exception {
            Class<?> clz = Class.forName("in.ashokit.BillCollector");
```

```
           Field field = clz.getDeclaredField("payment");
           field.setAccessible(true);

           Object obj = clz.newInstance();
           field.set(obj, new DebitcardPayment());//injecting value to
variable

           BillCollector bc = (BillCollector) obj;
           bc.collectPayment(2000.00);
     }
}
```

-> If we develop the project without using spring framework 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 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 colloborate 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.

Building First Application Using Spring Framework

----

1) Open STS IDE and Create Maven Project

2) Add Spring-Context dependency in project pom.xml file

```
<dependencies>
           <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>5.3.20</version>
           </dependency>
</dependencies>
```

3) Create required classes in our application using Strategy Design Pattern

```
            IPayment.java (I)
            CreditCardPayment.java
            DebitCardPayment.java
            UpiPayment.java
            BillCollector.java
            Test.java
```

4) Create Beans Configuration file and configure our classes as Spring Beans

```
      <bean id="creditCard" class="in.ashokit.CreditcardPayment" />

      <bean id="debitCard" class="in.ashokit.DebitcardPayment" />

      <bean id="upi" class="in.ashokit.UpiPayment" />

      <bean id="billCollector" class="in.ashokit.BillCollector">
            <property name="payment" ref="creditCard"/>
      </bean>
```

5) Start the IOC container and test the application

```
public class Test {

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

            ApplicationContext context = new
ClassPathXmlApplicationContext("Spring-Beans.xml");

            BillCollector bc = context.getBean("billCollector",
BillCollector.class);
            bc.collectPayment(1400.00);
      }
}
```

-> To perform setter injection we will use <property /> tag like below

```
<bean id="billCollector" class="in.ashokit.BillCollector">
            <property name="payment" ref="upi" />
</bean>
```

-> To perform constructor injection we will use <constructor-arg/> tag like bleow

```
<bean id="billCollector" class="in.ashokit.BillCollector">
            <constructor-arg name="payment" ref="upi" />
</bean>
```

-> When we perform both setter and constructor injection for same variable then setter injection will override constructor injection because construcor will execute first to initialize the variable then setter will execute and it will re-initialize the variable.

```
<bean id="billCollector" class="in.ashokit.BillCollector">
            <property name="payment" ref="creditCard" />
            <constructor-arg name="payment" ref="upi" />
```

```
</bean>
```

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

-> For prototype beans when we call context.getBean(..) method then object will be created

-> For prototype beans everytime new object will be created

-> To save memory spring framework made the default scope as singleton

```
      <bean id="motor" class="in.ashokit.Motor" scope="prototype" />

      <bean id="car" class="in.ashokit.Car" />
```

++
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="billCollector" class="in.ashokit.BillCollector">
            <constructor-arg name="payment" ref="upi" />
      </bean>
```

-> Using "ref" attribute we are telling to IOC which object it has to inject

=> This process is called as Manual Wiring

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

=> Autowiring having mode

1) byName
2) byType
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.

```
      <bean id="dieselEng" class="in.ashokit.beans.DieselEngine" />

      <bean id="car" class="in.ashokit.beans.Car" autowire="byName"/>
```

=> 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.

```
      <bean id="xyz" class="in.ashokit.beans.DieselEngine" />

      <bean id="car" class="in.ashokit.beans.Car" autowire="byType">
      </bean>
```

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

-> In byType mechanism if we have more than one bean matching with type then we will get ambiguity problem.

=> To overcome ambiguity problem we need to use 'autowire-candidate=false'

```
      <bean id="xyz" class="in.ashokit.beans.DieselEngine" autowire-
candidate="false"/>

      <bean id="abc" class="in.ashokit.beans.DieselEngine" />

      <bean id="car" class="in.ashokit.beans.Car" autowire="byType">
      </bean>
```

Note: When we configure autowiring with "byName" or "byType" it is performing setter injection by default and setter method is mandatory in target bean.

=> If we want to perfom Autowiring through constructor then we can use 'constructor' mode

```
      <bean id="xyz" class="in.ashokit.beans.DieselEngine" autowire-
candidate="false"/>

      <bean id="abc" class="in.ashokit.beans.DieselEngine" />
      <bean id="car" class="in.ashokit.beans.Car"
autowire="constructor"/>
```

=> In 'constructor' byType will be used to identify dependent bean object.

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
What is Spring Framework ?
Spring Modules
Tightly Coupling vs Loosely Coupling
Strategy Design Pattern
Spring Core Introduction
IOC Container
Dependency Injection
      - Setter Injection
      - Constructor Injection
      - Field Injection
<propery/> tag
<constructor-arg/> tag
Bean Scopes
      - singleton
      - prototype
<ref /> attribute
Autowiring
      - byName
      - byType
      - constructor
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Spring Boot & Microservices
+++++++++++++++++++++++++

-> Spring is an application development framework

-> By using spring framework we can develop below types of applications

```
                        1) standalone applications
                        2) web applications
                        3) distributed applications
```

-> Spring framework developed in modular fashion

-> Spring framework came into market in 2004

-> The current version of Spring is 5.x

-> Spring is non-invasive framework

-> Spring is versatile framework

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

-> After few years Spring team realized that for every project configurations are required

-> Every Programmer dealing with configurations in the project

-> The configurations are common in the project

=> To overcome configuration problems spring team released Spring Boot into market

+++++++++++++++
Spring Boot
++++++++++++++++

=> It is another approach to develop spring based applications with less configurations

=> Spring Boot is an enhacement 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 configurations required for our application and it will provide that configurations

```
                - Starting IOC container
                - Creating Connection Pool
                - Creation SMTP Connections
                - Web Application Deployment to server
                - Depedency 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 (default)
b) Jetty
c) Netty

-> Rapid Development means fast development. We can focus only on business logic because Boot will take care of configurations.

-> Actuators are used to monitor and manage our application. Actuators providing production-ready features for our application.

a) How many classes loaded
b) how many objects created
c) how many threads are running
d) URL Mappings Available
e) Health of the project etc...

+++++++++++++++++++++++++++++++++
Building First Spring Boot Application
+++++++++++++++++++++++++++++++++

-> We can create boot application in 2 ways

1) using Spring Initializr website (start.spring.io)

2) using IDE

Creating project using start.spring.io
++++++++++++++++++++++++++++++++

-> Goto start.spring.io website and generate the project

-> It will download project in zip file

-> Extract that zip file

-> Go t IDE -> New -> Import -> Maven -> Existring Maven Project -> Select Project path upto pom.xml file location

Creating project using STS IDE
++++++++++++++++++++++++

-> Go to STS IDE

-> New -> Spring Starter Project -> Fill the details and create the projet

Note: STS IDE will use start.spring.io internally to create the project

Note: TO create spring boot application internet connection is mandatory for our application.

```
Spring Boot Application Folder Structure
+++++++++++++++++++++++++++++++++++

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

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

      - src/main/resources
                 - application.properties or application.yml

      - 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 dependencies we will configure in pom.xml file

-> POM stands for Project Object Model


+++++++++++++++++++++++++++++++
What is start class in Spring Boot?
+++++++++++++++++++++++++++++++

-> 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

@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 (..) method contains bootstrapping logic. It is
the entry point for boot application execution

                 - start stop watch
                 - start listners
                 - prepareEnv
                 - print Banner
                 - create IOC container
                 - refresh Container
                 - stop that stop watch
                 - print time taken to start application
                 - call runners
                 - return IOC reference
```

++++++++++++++++++
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 name or project name as Banner text in Ascii format

Note: Generate Ascii text from here :
https://patorjk.com/software/taag/#p=display&f=Graffiti&t=Ashok%20IT

-> 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

```
            spring.main.banner-mode = off
```

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

```
@Component
public class CacheManager implements ApplicationRunner {

      @Override
      public void run(ApplicationArguments args) throws Exception {
            System.out.println("Logic executing to load data into
cache....");
      }
}

@Component
public class SendAppStartMail implements CommandLineRunner {

      @Override
      public void run(String... args) throws Exception {
            System.out.println("logic executing to send email....");
      }
}
```

Note: @Component annotation is used to represent our java class as Spring Bean

+++++++++++++++++++++++++++
What is @SpringBootApplication ?
++++++++++++++++++++++++++++

-> This annotation used at start class of spring boot

-> This annotation is equal to below 3 annotations

1) @SpringBootConfiguration
2) @EnableAutoConfiguration
3) @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.

```
                            in.ashokit     (base package)
                            in.ashokit.dao
                            in.ashokit.service
                            in.ashokit.config
                            in.ashokit.rest
                            in.ashokit.util
                            com.wallmart.security ----> This is not sub
package so scanning will not happen
```

-> We can specify more than one base package also in our boot start class

```
@SpringBootApplication
@ComponentScan(basePackages = { "in.ashokit", "com.wallmart" })
public class Application {

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

Note: It is highly recommended to follow proper package naming conventions

```
            Ex:   companyDomainName.projectName.moduleName.layerName

                  com.tcs.passport   (basePkgName)
                  com.tcs.passport.user.dao
                  com.tcs.passport.user.service
```

++++++++++++++++++++++
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?
+++++++++++++++++++++++++++++++++++++++

@Component
@Service
@Repository

@Controller
@RestController

@Configuration
@Bean

Note: 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.

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)

Note: Autowiring supports only referenced types (No suppot for primitive types)

Autowiring Example with @Qualifer
+++++++++++++++++++++++++++++++

-> When we use @Autowired annotation it will use byType mode to identify dependent object

-> If our interface having more than one impl then IOC will get confused to perform DI (Ambiguity)

-> To resolve that ambiguity problem we will use @Qualifier to speicify bean name to inject

Note: When we use @Qualifier it will use byName mode to inject dependent object

Note: If we don't want to use @Qualifier then we should specify @Primary for one bean to get injected

```
public interface ReportDao {
      public String findData();
}

@Repository("oracle")
public class OracleReportDaoImpl implements ReportDao {

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

      @Override
      public String findData() {
            System.out.println("fetching report data from oracle db...");
            return "Report data";
      }
}

@Repository("mysql")
public class MysqlReportDaoImpl implements ReportDao {

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

      @Override
      public String findData() {
            System.out.println("fetching report data from mysql db...");
            return "Report data";
      }
}

@Service
public class ReportService {

      private ReportDao reportDao;

      @Autowired
      @Qualifier("oracle")
      public void setReportDao(ReportDao reportDao) {
            System.out.println("setReportDao() method called...");
            this.reportDao = reportDao;
      }

      public void generateReport() {
            reportDao.findData();
            System.out.println("generating report....");
      }
}

@SpringBootApplication
public class Application {

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

            ReportService reportService =
context.getBean(ReportService.class);

            reportService.generateReport();
      }
}
```

@Autowired at Constructor Level
+++++++++++++++++++++++++++

-> If we have both 0-Param constructor and parameterized constructor then ioc will use 0-param constrictor to create object

-> If we want IOC to choose Param-Constructor to create obj then we should write @Autowired annotation at param-constructor

Note: If we have only param constructor in our class then @Autowired is optional

```
@Service
public class ReportService {

      private ReportDao reportDao;

      public ReportService(ReportDao reportDao) {
            System.out.println("ReportService :: Param Constructor
called...");
            this.reportDao = reportDao;
      }

      public void generateReport() {
            reportDao.findData();
            System.out.println("generating report....");
      }
}
```

@Autowired with Field Injection
++++++++++++++++++++++++

-> IOC will use Reflection api to internally to perform Field Injection

```
@Service
public class ReportService {

      @Autowired
      private ReportDao reportDao;

      public void generateReport() {
            reportDao.findData();
            System.out.println("generating report....");
      }
}
```

Note: Field Injection is not recommended because it is violating oops principles and it breaks Single Responsibility Principles.

-> When objects injected using field injection code review tools can't identify complexity.

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

Note: Out of all these Dependency Injections, Constructor Injection is recommended because Partial Injection not possible and target bean will be created if dependent bean is available.

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 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 bcz Data JPA will take care of that

-> Data JPA provideed Repository interfaces to perform CURD operations

1) CrudRepository (Methods to perform Crud Operations)

2) JpaRepository (Methods to perform Crud Operations + Pagination + Sorting + QBE)

Hibernate Vs Data JPA
++++++++++++++++++

-> In Hibernate we should implement all methods to perform CRUD operations
-> Data JPA providing predefined methods to perform CRUD Operations

-> In Hibernate we should boiler plate code (same code in multiple classes)
-> In Data JPA we don't need to write any method because JPA Repositories providing methods for us

Environment Setup
+++++++++++++++++

1) MySQL Database (Server s/w)
(https://dev.mysql.com/downloads/installer/)
2) MySQL Workbench (Client s/w)
(https://dev.mysql.com/downloads/workbench/)

MySQL DB Properties
++++++++++++++++++

```
spring.datasource.url=jdbc:mysql://localhost:3306/sbms
spring.datasource.username=ashokit
spring.datasource.password=AshokIT@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

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

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 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;
}
```

Note: One entity class will be mapped with only one DB table.

+++++++++++++++++++
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.

++++++++++++++++++++++
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.url=jdbc:mysql://localhost:3306/sbms
spring.datasource.username=ashokit
spring.datasource.password=AshokIT@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

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

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

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 single 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) existsById (.) : 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 with given entities

12) deleteAll ( ) : To delete all records

```
@Entity
@Table(name = "USER_MASTER")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

      @Id
      @Column(name = "USER_ID")
      private Integer userid;

      @Column(name = "USER_NAME")
```

```
	private String username;

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

	@Column(name = "USER_AGE")
	private Integer age;

	@Column(name = "USER_COUNTRY")
	private String country;

}
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-------------------
public interface UserRepository extends CrudRepository<User, Integer> {

}
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-------------------

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

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

		/*User u1 = new User(101, "Ramu", "Male", 25, "India");

		repository.save(u1);*/

		/*User u2 = new User(102, "Raju", "Male", 26, "India");
		User u3 = new User(103, "John", "Male", 30, "USA");
		User u4 = new User(104, "Smith", "Male", 32, "Canada");

		repository.saveAll(Arrays.asList(u2, u3, u4));*/

		/*Optional<User> findById = repository.findById(103);
		if(findById.isPresent()) {
			System.out.println(findById.get());
		}*/

		/*Iterable<User> allById =
repository.findAllById(Arrays.asList(101,102,103));
		allById.forEach(user -> {
			System.out.println(user);
		});*/

		/*Iterable<User> findAll = repository.findAll();
		findAll.forEach(user -> {
			System.out.println(user);
		});*/

		/*long count = repository.count();
		System.out.println("Total Records in table :: "+ count);*/
```

```
			/*boolean existsById = repository.existsById(101);
			System.out.println("Record Presence with id - 101 :: " +
existsById);*/

			//repository.deleteById(104);

			repository.deleteAllById(Arrays.asList(102,103));
		}
}
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/sbms
spring.datasource.username=ashokit
spring.datasource.password=AshokIT@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver


spring.jpa.hibernate.ddl-auto=update

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

-> 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

	SQL : select * from user_master where user_country='INDIA';

Requirement -2 : Retrieve users whose age is below 30 years

	SQL : select * from user_master where user_age >=30;

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 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.

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

-> 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)

Note: In Hibernate we have Dialect classes for every database

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

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

# Retrieve all records from table

SQL : SELECT * FROM USER_MASTER

HQL : From User

# Retrieve all records of users who are belongs to country 'India'

SQL : SELECT * FROM USER_MASTER WHERE USER_COUNTRY = 'INDIA'

HQL : From User where country='India'

# Retrieve users who are belongs to 'India' and age is 25

SQL : SELECT * FROM USER_MASTER where USER_COUNTRY='INDIA' AND USER_AGE = 25

HQL : From User where country='India' and age = 25

# Retrieve User ID and User Name based on User Country

SQL : select user_id, user_name from User_master where user_country='India';

HQL : select userid, username from User where country='India'

Note: If we don't have projection then we can start HQL query with 'FROM' keyword. If we have projection then we need to start with 'SELECT' keyword

Note: Projection means retriving specific columns data from the table

-> To write custom queries we will use @Query annotation in Repository interface

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++
public interface UserRepository extends CrudRepository<User, Integer> {

      @Query(value = "From User")
      public List<User> getAllUsersHql();

      @Query(value = "select * from user_master", nativeQuery = true)
      public List<User> getAllUsersSql();

      @Query(value = "From User where country=:cname")
      public List<User> getAllUsersByCountry(String cname);

      @Query(value = "From User where country=:cname and age=:age")
      public List<User> getAllUsersByCountryAndAge(String cname, Integer
age);

}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++
```

We can perform DB operations using Data JPA in below 3 ways

1) Predefined methods

2) findByXXX methods

3) Custom Queries (@ Query)

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

Sorting
++++++++
-> Sorting is used to sort the records either in ascending or in descending order

-> We can pass Sort object as parameter for findAll ( ) method like below

```
		List<User> users = repository.findAll();
		List<User> users = 
repository.findAll(Sort.by("age").ascending());
		List<User> users = 
repository.findAll(Sort.by("username","age").descending());

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

++++++++++
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)

```
		int pageSize = 5;
		int pageNo  = 1;

		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)

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

-> 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);
            });



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

++++++++++++++++++++++++++++++++Example+++++++++++++++++++++++++++++++++
+++++++++++++++++++

@Data
@Entity
@Table(name = "PRODUCT_MASTER")
public class Product {

      @Id
```

```
	@Column(name = "PRODUCT_ID")
	private Integer pid;

	@Column(name = "PRODUCT_NAME")
	private String pname;

	@Column(name = "PRODUCT_PRICE")
	private Double price;

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

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

}
```

-> 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.

+++

Q) Why to use Wrapper classes instead of Primitive datatypes in Entity class ?

-> Primitive data types will take default values when we dont set the value for variable

Ex : if we take 'price variable with double data type' then it will insert price as '0' if we don't set price for the record. It is not recommended.

-> If we take wrapper classes it will consider 'null' as default value when we don't set value for a variable.

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

-> 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

Ex:

```
create sequence product_id_seq
start with 101
increment by 1;
```

-> If we want to generate primary key column value like below then we should go for Custom Generator (Our own Generator we have to develop)

```
TCS01
TCS02
TCS03
TCS04
```

-> Custom Generator Example : https://youtu.be/IijGVtT9ZPk

+++++++++++++++++++++
Composite Primary Keys
++++++++++++++++++++

-> Table can have more than one primary key column

-> If table contains more than one primary key column then those primary keys are called as Composite Primary Keys

```
CREATE TABLE ACCOUNTS
(
   ACC_ID             NUMBER,
   ACC_NUMBER         NUMBER,
   HOLDER_NAME        VARCHAR2(50)
   ACC_TYPE                 VARCHAR2(10),
   BRANCH_NAME        VARCHAR2(10)

   PRIMARY KEY (ACC_ID, ACC_NUMBER, BRANCH_NAME)

)
```

Note: For Composite Primary Keys we can't use Generator. We have to set values manually

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

	private Integer accId;
	private String accType;
	private String holderName;
}

@Data
@Entity
@Table(name = "BANK_ACCOUNTS")
public class Account {

	@Column(name = "BRANCH_NAME")
	private String branchName;

	@Column(name = "MIN_BAL")
	private Double minBal;

	@EmbeddedId
	private AccountPK accPk;

}

public interface AccountRepository extends JpaRepository<Account,
AccountPK> {

}

@Service
public class AccountService {

	private AccountRepository accRepo;

	public AccountService(AccountRepository accRepo) {
		this.accRepo = accRepo;
	}

	public void getDataUsingPK() {

		AccountPK pk = new AccountPK();
		pk.setAccId(101);
		pk.setAccType("CURRENT");
		pk.setHolderName("IBM");

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

	public void saveAccData() {
		AccountPK pk = new AccountPK();
		pk.setAccId(104);
		pk.setAccType("SAVINGS");
		pk.setHolderName("TCS");

		Account acc = new Account();
		acc.setBranchName("Ameerpet");
```

```
            acc.setMinBal(5000.00);

            acc.setAccPk(pk); // setting pk class obj to entity obj

            accRepo.save(acc);
      }
}


@SpringBootApplication
public class Application {

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

            AccountService accountService =
context.getBean(AccountService.class);
            accountService.saveAccData();
            accountService.getDataUsingPK();

            context.close();
      }
}
```

++++++++++++++++++++++++
TX 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

-> COMMIT means storing data in database permanently

-> ROLLBACK means bringing database to previous state

```
+++++++++++++++++++++++++Example for
Rollback++++++++++++++++++++++++++++++

      @Transactional(rollbackFor = Exception.class)
      public void saveData() {

            Employee emp = new Employee();
            emp.setEmpId(201);
            emp.setEmpName("Ketan");
            emp.setEmpSal(25000.00);
            empRepo.save(emp);

            int i = 10 / 0;

            Address addr = new Address();
            addr.setAddrId(501);
            addr.setEmpId(201);
            addr.setCity("Pune");
            addr.setState("MH");
            addr.setCountry("India");
            addrRepo.save(addr);
      }

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++
Requirement : Develop Data JPA Application to insert image file into
database table
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++

USER_TBL (TABLE NAME)
-------------------------------------

USER_ID               INTEGER          PRIMARY KEY AUTO_INCREMENT

USER_NAME        VARCHAR

USER_EMAIL       VARCHAR

USER_PHOTO  BLOB




@Entity
@Table(name = "USER_TBL")
@Data
public class User {

      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      @Column(name = "USER_ID")
      private Integer userId;

      @Column(name = "USER_NAME")
      private String userName;

      @Column(name = "USER_EMAIL")
      private String userEmail;
      @Column(name = "USER_IMAGE")
      @Lob
      private byte[] userImage;

}

public interface UserRepository extends JpaRepository<User, Integer> {

}

@Service
public class UserService {

      @Autowired
      private UserRepository userRepo;

      public void saveUser() throws Exception {

            String imagePath = "give-file-path";

            User user = new User();
            user.setUserName("Suresh");
            user.setUserEmail("suresh@gmail.com");

            long size = Files.size(Paths.get(imagePath));

            byte[] arr = new byte[(int) size];
            FileInputStream fis = new FileInputStream(new
File(imagePath));
            fis.read(arr);
            fis.close();

            user.setUserImage(arr);

            userRepo.save(user);
      }
}
```

++++++++++++++
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 (Webservices)

-> 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
-> Distributed Applications we can develop in 2 ways

1) SOAP Webservices
2) RESTFul Services

Note: SOAP Webservices & RESTFul Services can be developed using Spring Web MVC

-> Distributed applications we are developing to re-use logic of one application in another application.

Ex:

```
MakeMyTrip  -----------> IRCTC
Passport  -----------> AADHAR
Gpay ---------------> Banking Apps
Swiggy  ----------> Banking Apps
```

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

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

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 with prefix and suffix

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

# tomact-embed-jasper dependency

```
<dependency>
      <groupId>org.apache.tomcat.embed</groupId>
      <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

# Controller class

```
@Controller
public class WelcomeController {

      @GetMapping("/welcome")
      public ModelAndView getWelcomeMsg() {

            ModelAndView mav = new ModelAndView();

            mav.addObject("msg", "Welcome to Ashok IT...!!");

            mav.setViewName("index");

            return mav;
      }
}
```

# view resolver configuration in application.properties

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

# jsp file

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
      <h1>${msg}</h1>
</body>
</html>
```

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

Note: Embedded Tomcat Server will run on the port number 8080. This is default behaviour.

-> We can change embedded server port number using below property in application.properties

```
                  server.port=9090
```

++++++++++++++++++++++++++++++++++
Sending Data From Controller To UI
+++++++++++++++++++++++++++++++++++

-> We can send data from controller to UI in multiple ways

1) ModelAndView

2) Model

3) @ResponseBody

################# Approach-1   (ModelAndView) #####################

```
@Controller
public class WelcomeController {

      @GetMapping("/welcome")
      public ModelAndView welcomeMsg() {

            ModelAndView mav = new ModelAndView();

            mav.addObject("msg", "Welcome to Ashok IT");

            mav.setViewName("welcome");

            return mav;
      }
}
```

################# Approach-2   (Model) #####################

```
@Controller
public class GreetController {
	@GetMapping("/greet")
	public String getGreetMsg(Model model) {

		String msgTxt = "Good Morning..";

		model.addAttribute("msg", msgTxt);

		return "greet";
	}
}


################# Approach-3   ( @ResponseBody ) #####################

@Controller
public class WishController {

	@GetMapping("/wish")
	@ResponseBody
	public String getWishMsg() {

		String msg = "All the best...!!!";

		return msg;
	}
}



################# sending object data from Controller to UI
#######################

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Book {

	private Integer bookId;
	private String bookName;
	private Double bookPrice;

}

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

@Controller
public class BookController {

	@GetMapping("/book")
	public String getBookData(Model model) {

		// setting data to binding obj
		Book bookObj = new Book(101, "Spring", 450.00);

		// adding data to model obj to send to UI
		model.addAttribute("book", bookObj);

		// return view name
		return "book";
	}
```

```
}

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

<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

      <h2>Book Data</h2>

      Book Id : ${book.bookId} <br/>

      Book Name : ${book.bookName} <br/>

      Book Price : ${book.bookPrice} <br/>

</body>
</html>

--------------------------------------------
```

Assignment: Develop Spring Boot web application to display multiple books in a table format.

```
            <dependency>
                  <groupId>javax.servlet</groupId>
                  <artifactId>jstl</artifactId>
            </dependency>

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

      @GetMapping("/books")
      public String getBooksData(Model model) {
            // setting data to binding obj
            Book b1 = new Book(101, "Spring", 350.00);
            Book b2 = new Book(102, "Python", 450.00);
            Book b3 = new Book(103, "AWS", 550.00);

            List<Book> booksList = Arrays.asList(b1, b2, b3);

            // adding data to model obj to send to UI
            model.addAttribute("books", booksList);

            // return view name
            return "books";
      }
-----------------------------------------

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<table border="1">
		<thead>
			<tr>
				<th>Book ID</th>
				<th>Book Name</th>
				<th>Book Price</th>
			</tr>
		</thead>
		<tbody>
			<c:forEach items="${books}" var="book">
				<tr>
					<td>${book.bookId}</td>
					<td>${book.bookName}</td>
					<td>${book.bookPrice}</td>
				</tr>
			</c:forEach>
		</tbody>
	</table>
</body>
</html>
```

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

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)

-> To work with Spring Form Tag library we need to use below taglib directive

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

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

1) Create boot app with below dependencies

    a) web-starter
    b) devtools
    c) lombok
    d) tomcat-embed-jasper

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

5) configure view resolver in application.properties file

```
@Data
public class Product {

      private Integer productId;

      private String productName;

      private Double productPrice;

}
```

```
@Controller
public class ProductController {

      @GetMapping("/")
      public String getProductForm(Model model) {
            Product productObj = new Product();
            model.addAttribute("product", productObj);
            return "index";
```

```
	}

	@PostMapping("/product")
	public String handleFormSubmit(Product product, Model model) {
		System.out.println(product);
		model.addAttribute("msg", "Product Saved Successfully");
		return "success";
	}
}
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>

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

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

	<h3>Save Product Data</h3>
	<form:form action="product" modelAttribute="product" method="POST">
		<table>
			<tr>
				<td>Product ID</td>
				<td><form:input path="productId" /></td>
			</tr>
			<tr>
				<td>Product Name</td>
				<td><form:input path="productName" /></td>
			</tr>
			<tr>
				<td>Product Price</td>
				<td><form:input path="productPrice" /></td>
			</tr>
			<tr>
				<td><input type="submit" value="Submit" /></td>
			</tr>
		</table>
	</form:form>
</body>
</html>
```


```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
	<h2>${msg}</h2>
```

```
      <a href="/">Go Home</a>
</body>
</html>
```

+++++++++++++++++++
Form Validations
+++++++++++++++++++

-> Forms are used to capture data from the user

-> To make sure users are entering valid data we will form validations on the data

-> Spring Web MVC having support to perform form validations....

@NotEmpty

@NotNull

@Size

@Min

@Max

-> We need to below dependency in pom.xml to perform form validations

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

```
@Data
public class User {

      @NotEmpty(message = "Uname is required")
      @Size(min = 3, max = 8, message = "Uname should be 3 to 8 
characters")
      private String uname;

      @NotEmpty(message = "Pwd is required")
      private String pwd;

      @NotEmpty(message = "Email is required")
      @Email(message = "Enter valid email id")
      private String email;

      @NotEmpty(message = "Phno is required")
      @Size(min = 10, message = "Phno should have atleast 10 digits")
      private String phno;

      @NotNull(message = "Age is required")
      @Min(value = 21, message = "Age should be minimum 21 years")
      @Max(value = 60, message = "Age shouldn't cross 60 years")
      private Integer age;

}

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

@Controller
public class UserController {

      @GetMapping("/")
      public String getForm(Model model) {
            User userObj = new User();
            model.addAttribute("user", userObj);
            return "index";
      }

      @PostMapping("/register")
      public String handleRegisterBtn(@Valid User userForm, BindingResult
result, Model model) {
            if(result.hasErrors()) {
                  return "index";
            }
            System.out.println(userForm);
            //logic to store form data in db
            model.addAttribute("msg", "Your Registration
Successful...!!");
            return "success";
      }
}
--------------------------------------
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>

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

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>

<style>
.error {
      color: red
}
</style>

</head>
<body>

      <h3>User Registration Form</h3>
      <form:form action="register" modelAttribute="user" method="POST">
            <table>
                  <tr>
```

```
                        <td>Username</td>
                        <td><form:input path="uname" /> <form:errors
path="uname" cssClass="error"/></td>
                  </tr>
                  <tr>
                        <td>Pwd</td>
                        <td><form:password path="pwd" /> <form:errors
path="pwd" cssClass="error"/></td>
                  </tr>
                  <tr>
                        <td>Email</td>
                        <td><form:input path="email" /> <form:errors
path="email" cssClass="error"/></td>
                  </tr>
                  <tr>
                        <td>Phno</td>
                        <td><form:input path="phno" /> <form:errors
path="phno" cssClass="error"/> </td>
                  </tr>
                  <tr>
                        <td>Age</td>
                        <td><form:input path="age" /> <form:errors
path="age" cssClass="error"/> </td>
                  </tr>
                  <tr>
                        <td></td>
                        <td><input type="submit" value="Register" /></td>
                  </tr>
            </table>
      </form:form>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
      <h2>${msg}</h2>

      <a href="/">Home</a>

</body>
</html>
```

+++++++++
Thymeleaf
+++++++++++

-> We used JSP as a presentation technology in our spring web mvc based applications

-> JSP can't be executed in browser directley

-> When the request comes to JSP then internally JSP will be converted to Servlet and that servlet will send response to browser

-> When we use JSP for presentation then burden will be increased on server because every JSP should be converted into Servlet to produce the response to browser.

-> To overcome problems of JSP we can use Thymeleaf as a presentation technology

-> Thymleaf is a template engine that can be used in HTML pages directley

-> HTML pages can be executed in browser directley
  (Thymeleaf performance will be fast when compared with jsps)

-> In general, HTML pages are used for static data. If we use thymleaf in HTML then we can add dynamic nature to HTML pages.

-> We can develop spring boot application with thymleaf as a presentation technology

-> To use Thymleaf in spring boot we have below starter

'spring-boot-starter-thymleaf'

Procedure to develop spring boot application with thymeleaf
----

1) Create Spring Starter Project with below dependencies

    a) web-starter
    b) thymeleaf-starter
    c) devtools

2) Create Controller with required methods (@Controller)

3) Create Theymeleaf templates in src/main/resources/templates folder (file extension .html)

4) Run the application and test it

Note: No need to configure view resolver because Spring Boot will detect theymeleaf template files and will process them

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

```
@Controller
public class WelcomeController {

      @GetMapping("/welcome")
      public String welcomeMsg(@RequestParam("name") String name, Model
model) {
            String msgTxt = name + ", Welcome to Ashok IT..!!";
            model.addAttribute("msg", msgTxt);
            return "index";
      }
}
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

      <p th:text=${msg} />

</body>
</html>
```

Spring Boot + Thymeleaf (Form Based Application)
----

1) create a spring starter project with below dependencies

   a) web-starter
   b) thymeleaf-starter
   c) lombok
   d) devtools

2) Create Form Binding Class

```
@Data
public class Product {
	private Integer pid;
	private String pname;
	private Double price;

}
```

3) Create Controller

```
@Controller
public class ProductController {

	@GetMapping("/product")
	public ModelAndView loadForm() {
		ModelAndView mav = new ModelAndView();
		mav.addObject("product", new Product());
		mav.setViewName("productView");
		return mav;
	}

	@PostMapping("/product")
	public ModelAndView handleSubmitBtn(Product product) {
		ModelAndView mav = new ModelAndView();
		mav.setViewName("successView");
		return mav;
	}
}
```

4) Develop view file display form (productView.html)

```
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

	<form th:action="@{/product}" th:object="${product}" method="POST">
		<table>
			<tr>
				<td>Product Id:</td>
				<td><input type="text" th:field="*{pid}" /></td>
			</tr>
			<tr>
				<td>Product Name:</td>
				<td><input type="text" th:field="*{pname}" /></td>
			</tr>
			<tr>
				<td>Product Price:</td>
				<td><input type="text" th:field="*{price}" /></td>
			</tr>

			<tr>
				<td></td>
				<td><input type="submit" value="Save" /></td>
			</tr>
		</table>
	</form>
</body>
```

```
</html>
```

5) Develop view file to display success message (successView.html)

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
      <h1> Product Record Saved Successfully</h1>
      <a href="product">Go Back</a>
</body>
</html>
```

6) Configure the port number and run the application.

==
Spring Boot + Thymeleaf + Form validations - Example

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.6.2</version>
            <relativePath/> <!-- lookup parent from repository -->
      </parent>
      <groupId>in.ashokit</groupId>
      <artifactId>25-SB-Web-MVC-Form-Validations</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <name>25-SB-Web-MVC-Form-Validations</name>
      <description>Demo project for Spring Boot</description>
      <properties>
            <java.version>1.8</java.version>
      </properties>
      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-thymeleaf</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-validation</artifactId>
            </dependency>
            <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>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
            </dependency>
        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
            </plugins>
        </build>

</project>
```

```
---------------------------------------------------------------------------
----------------------
package in.ashokit.binding;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Person {

      @NotNull
      @Size(min = 3, max = 8)
      private String name;

      @NotNull
      @Min(18)
      private Integer age;

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

      public Integer getAge() {
            return age;
      }

      public void setAge(Integer age) {
            this.age = age;
      }

      @Override
      public String toString() {
            return "Person [name=" + name + ", age=" + age + "]";
      }
}
---------------------------------------------------------------------------
----------------------
package in.ashokit.controller;

import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

import in.ashokit.binding.Person;

@Controller
public class PersonController {

	@GetMapping("/person")
	public String displayForm(Model model) {
		Person personObj = new Person();
		model.addAttribute("person", personObj);
		return "index";
	}

	@PostMapping("/savePerson")
	public String savePerson(@Valid Person person, BindingResult 
result, Model model) {
		System.out.println(person);

		if (result.hasErrors()) {
			return "index";
		}

		model.addAttribute("msg", person.getName() + " record saved 
successfully");
		return "data";
	}

}
---------------------------------------------------------------------------
----------------------
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

	<form th:action="@{/savePerson}" th:object="${person}" 
method="POST">
		<table>
			<tr>
				<td>Name :</td>
				<td><input type="text" th:field="*{name}" /></td>

			</tr>
			<tr>
```

```
                        <td>Age :</td>
                        <td><input type="text" th:field="*{age}" /></td>

                  </tr>
                  <tr>
                        <td></td>
                        <td><input type="submit" value="Save" /></td>
                  </tr>
            </table>
      </form>

</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
      <p th:text="${msg}" />

      <a href="person">Go Back</a>

</body>
</html>
```

++++++++++++++++++
Embedded Servers
++++++++++++++++++

-> Spring Boot provided embedded containers to run our web applications

-> When we add "web-starter" by default it is giving 'tomcat' as default embedded container

-> In spring boot we have multiple embedded containers

1) tomcat
2) jetty
3) netty
4) undertow etc...

Note: we can deploy spring boot application into external servers also as a war file.

Q) How change default container from tomcat to jetty ?

-> To make jetty as embedded container we need make below 3 changes in pom.xml file

1) Remove tomcat-starter dependency

2) Exclude tomcat-starter from web-starter

3) Add jetty dependency

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

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

----

1) What is Spring Web MVC ?
2) What is C2B & B2B ?
3) Spring Web MVC advantages
4) Spring Web MVC architecture
5) Front Controller (DispatcherServlet)
6) HandlerMapper
7) Controller
8) ViewResolver
9) View
10) Web MVC Annotations
    - @Controller
    - @GetMapping
    - @PostMapping

11) ModelAndView
12) Model
13) Sending data from Controller to UI
14) Spring MVC Form Tag Library
15) Forms Development
16) Form Validations
17) Thymeleaf Introduction
18) Form development using Thymleaf

++
50 Interview Questions on Spring Web MVC :
++

https://www.youtube.com/watch?v=1_SsosC4Cs8&list=PLpLBSl8eY8jSMr1hJLB096n
q8W0ABQoXH

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 B 2 B 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 ?
################

-> 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 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

a) GET
b) POST
c) PUT
d) 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 data 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
-> PUT request contains request body
-> PUT request is Idempotent

Note: In PUT request we can send data 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 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 (Protocl Version + Status Code + Status msg)
2) Response Headers (Key-value)
3) Blank Line to seperate Header & Body
4) Response Body (Response Payload)

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

-> HTTP Status codes will represent how the request process by server / provider

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

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

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

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

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

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 the XML

1) Simple Element

2) Compound Element

-> The element which contains data directley is called as Simple Element

```
           <name> Ashok IT </name>

           <type> Educational </type>
```

-> The element which contains child element(s) is called as Compound Element

```
            <person>
                  <id> 101 </id>
                  <name> raju </name>
            </person>
```

Note: here <person> is a compound element and <id> <name> are simple elements

-> We can have attributes also for the element

```
            <student  branch="CSE">
                  <id> 101 </id>
                  <name> Mahesh </name>
            </student>
```

Note: XML should have only one root element. Inside the root element we can have multiple child elements

```
<persons>
            <person>
                  <id> 101 </id>
                  <name> raju </name>
            </person>

            <person>
                  <id> 101 </id>
                  <name> raju </name>
            </person>
</persons>
```

###########
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.

#####################
Marshalling Example
#####################

```
@Data
@XmlRootElement
public class Person {

      private Integer id;
      private String name;
      private Integer age;
      private Long phno;
      private Address adress;
}
----------------------------------
@Data
public class Address {

      private String city;
      private String state;
      private String country;
}
----------------------------------------
public class ConverJavaToXml {

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

            Address addr = new Address();
            addr.setCity("Hyd");
            addr.setState("TG");
            addr.setCountry("India");

            Person person = new Person();
            person.setId(101);
            person.setName("John");
            person.setAge(25);
            person.setPhno(12575757l);
            person.setAdress(addr);

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

            Marshaller marshaller = instance.createMarshaller();

            marshaller.marshal(person, new File("Person.xml"));

            System.out.println("Marshalling Completed....");
      }
}
---------------------------------------------------------------------------
----
```

@XmlAccessorType(XmlAccessType.FIELD) : Controls marshalling and un-marshalling using fields of entity class

@XmlAccessorOrder : Follow order of variables in the class to marshall and un-marshall

@XmlElement(name = "PhoneNum") : It is used to change the name of element

@XmlAttribute : It represents variable as attribute in xml

@XmlTransient : To skip a variable in marshalling

Note: By default every variable will be considered as Element and variable name will be considered as element name.

```
@Data
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@XmlAccessorOrder
public class Person {

      private Integer id;
      private String name;

      @XmlTransient
      private Integer age;

      @XmlElement(name = "PhoneNum")
      private Long phno;

      @XmlAttribute
      private String type;

      private Address adress;
}
---------------------------------------------------------------------------
--------------------------

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 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

-> To work with JSON data in Java Applications we have below 3rd party APIs

1) JACKSON API

2) GSON API

-> By using above apis we can convert JSON data to Java Object and vice versa

-> The process of converting Java Object into JSON is called as Serialization

-> The process of converting JSON data to Java Object is called as De-Serialization

1) Create Maven project with below dependencies

```
	<dependencies>
		<dependency>
			<groupId>com.fasterxml.jackson.core</groupId>
			<artifactId>jackson-databind</artifactId>
			<version>2.13.3</version>
		</dependency>
		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<version>1.18.24</version>
			<scope>provided</scope>
		</dependency>
	</dependencies>
```

2) Create Java classes to represent data (Use lombok)

```
@Data
public class Author {

	private String authorName;
	private String authorEmail;
	private Long authorPhno;

}

@Data
public class Book {

	private Integer id;
	private String name;
	private Double price;
	private Author author;
}
```

3) Create Java class to convert Java Obj to JSON file

```
public class JavaToJsonConverter {

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

		Author author = new Author();
		author.setAuthorName("Rod Johnson");
		author.setAuthorEmail("r.john@gmail.com");
		author.setAuthorPhno(868686861);

		Book book = new Book();
		book.setId(101);
		book.setName("Spring");
		book.setPrice(450.00);
		book.setAuthor(author);
```

```
            ObjectMapper mapper = new ObjectMapper();

            // converting java obj to json and store into a file
            mapper.writeValue(new File("book.json"), book);

            System.out.println("Conversion Completed....");
      }
}
```

4) Create Java Class To Convert JSON to Java Object

```
public class JsonToJavaConverter {

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

            File jsonFile = new File("book.json");

            ObjectMapper mapper = new ObjectMapper();

            Book book = mapper.readValue(jsonFile, Book.class);

            System.out.println(book);

      }
}
```

++++++++++++++++++++++
Working with GSON API
+++++++++++++++++++++++

1) Create a maven project with below dependency

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.9.0</version>
</dependency>
```

-> GSON api provided by google

-> In GSON api we have 'Gson' class to perform conversions

            toJson ( ) -> to convert java object to JSON

            fromJson ( ) -> to convert json data to java object

What is Disributed Application
What is Intereoperability
What is HTTP Protocol
HTTP Methods
HTTP Status Codes
HTTP Request Structure
HTTP Response Structure
Working with XML and JAX-B

Working with JSON and Jackson

```
+++++++++++++++++++++++++++++++
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: RESTful Services are used to develop B2B communications (No presentation logic, No view Resolver)

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

```
+++++++++++++++++++++++++++++++++++
@RestController
public class WelcomeRestController {

      @GetMapping("/welcome")
      public ResponseEntity<String> getWelcomeMsg() {
            String respPayload = "Welcome to Ashok IT";
            return new ResponseEntity<>(respPayload, HttpStatus.OK);
      }

      @GetMapping("/greet")
      public String getGreetMsg() {
            String respPayload = "Good Morning..!!";
            return respPayload;
      }
}
+++++++++++++++++++++++++++++++++++
```

Note: GET Request will not contain Request Body to send data

-> We can use Query Params and Path Params to send data in GET Request

-> Query Params & Path Params will represent data in URL directlry

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

```
	@GetMapping("/welcome")
	public ResponseEntity<String> getWelcomeMsg(@RequestParam("name") 
String name) {
		String respPayload = name + ", Welcome to Ashok IT";
		return new ResponseEntity<>(respPayload, HttpStatus.OK);
	}

	URL : http://localhost:8080/welcome?name=Raju
```

++++++++++++++++++++++++++++++++++
Working with 2 Query Params in URL
++++++++++++++++++++++++++++++++++++

```
@RestController
public class CourseRestController {

	@GetMapping("/course")
	public ResponseEntity<String> getCourseFee(@RequestParam("cname") 
String cname,
			@RequestParam("tname") String tname) {

		String respBody = cname + " By " + tname + " Fee is 7000 INR";

		return new ResponseEntity<>(respBody, HttpStatus.OK);

	}
}
```

URL : http://localhost:8080/course?cname=JRTP&tname=Ashok

++++++++++++++++++++++++++++
Path Parameter or URI variables
+++++++++++++++++++++++++++++

-> 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)

```
		Ex:  www.ashokit.in/courses/{cname}/trainer/{tname}
```

-> To read Path Parameters we will use @PathVariable annotation

```
@RestController
public class BookRestController {

      @GetMapping("/book/{name}")
      public ResponseEntity<String> getBookPrice(@PathVariable("name")
String name) {

            String respBody = name + " Price is 400 $";

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

      @GetMapping("/book/name/{bname}/author/{aname}")
      public ResponseEntity<String> getBook(@PathVariable("bname") String
bname,
                  @PathVariable("aname") String aname) {

            String respBody = bname + " By " + aname + " is out of stock";

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

URL-1 : http://localhost:8080/book/spring

URL-2 : http://localhost:8080/book/name/spring/author/rodjohnson

+++++++++++++++++++++++++++++++++++++++
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
#################

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

```
            produces = { "application/xml", "application/json" }
```

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

```
@Data
@XmlRootElement
@NoArgsConstructor
@AllArgsConstructor
```

```
public class Product {

      private Integer pid;
      private String pname;
      private Double price;

}

@RestController
public class ProductRestController {

      @GetMapping(
                  value = "/product",
                  produces = { "application/xml", "application/json" }
      )
      public ResponseEntity<Product> getProduct() {

            Product p1 = new Product(101, "Monitor", 8000.00);

            return new ResponseEntity<>(p1, HttpStatus.OK);
      }

      @GetMapping("/products")
      public ResponseEntity<List<Product>> getProducts(){

            Product p1 = new Product(101, "Monitor", 8000.00);
            Product p2 = new Product(102, "RAM", 6000.00);
            Product p3 = new Product(103, "CPU", 15000.00);

            List<Product> products = Arrays.asList(p1,p2,p3);

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

##############################
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
@XmlRootElement
```

```
public class Book {

      private Integer id;
      private String name;
      private Double price;

}
---------------------------
@RestController
public class BookRestController {

      @PostMapping(
                  value = "/book",
                  consumes = { "application/json", "application/xml" }
      )
      public ResponseEntity<String> addBook(@RequestBody Book book) {
            System.out.println(book);

            // logic to store in DB

            String msg = "Book Added Succesfully";

            return new ResponseEntity<String>(msg, HttpStatus.CREATED);
      }
}

----------------
{
      "id" : 101,
      "name" : "Java",
      "price" : 450.00
}

------------------------
```

-> produces vs consumes

-> Content-Type vs Accept

-> produces attribute represents in which formats Method can provide response data to clients

-> consumes attribute represents in which formats Method can take request data from clients

-> Accept header represents in which format client expecting response from REST API

-> Content-Type header represents in which format client is sending request data to REST API

Note: We can use both Consumes & Produces in single REST API method.

++
Requirement : Develop IRCTC REST API to book a train ticket
++

-> To develop any REST API first we have to understand the requirement

-> Identify input / request data

-> Identify output / response data

-> Create request & response binding classes

-> Create REST Controller with required methods.

-> Test REST API methods behaviour using Postman

```
@Data
public class PassengerInfo {

      private String name;
      private Long phno;
      private String jdate;
      private String from;
      private String to;
      private Integer trainNum;

}

@Data
public class TicketInfo {

      private Integer ticketId;
      private String pnr;
      private String ticketStatus;

}

@RestController
public class TicketRestController {

      @PostMapping(
                  value = "/ticket",
                  produces = {"application/json"},
                  consumes = {"application/json"}
      )
      public ResponseEntity<TicketInfo> bookTicket(@RequestBody
PassengerInfo request){
            System.out.println(request);

            //logic to book ticket
            TicketInfo tinfo = new TicketInfo();
            tinfo.setTicketId(1234);
            tinfo.setPnr("JLJL6868");
            tinfo.setTicketStatus("CONFIRMED");

            return new ResponseEntity<>(tinfo, HttpStatus.CREATED);
      }
}
```

```
{
      "name"   : "Ashok",
      "phno"      : 12345678,
      "jdate"  : "05-08-2022",
      "from"   : "hyd",
      "to"     : "pune",
      "trainNum" : 8574
}

{
    "ticketId": 1234,
    "pnr": "JLJL6868",
    "ticketStatus": "CONFIRMED"
}
```

#################
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);
      }
```

####################
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);
      }
```

What is RestController ?
REST Controller Methods
@GetMapping
@PostMapping
@PutMapping

@DeleteMapping

Query Params
Path Params
Request Body
Response Body

@RequestParam
@PathVariable
@RequestBody

produces
consumes
Accept
Content-Type

Message Converters

ResponseEntity

Q) Can we write the logic to update a record in POST request method ?

Ans) Yes, we can do it but not recommended. We need to follow HTTP Protocol standards while developing REST API.

+++++++++
Swagger
+++++++

-> Swagger is used to generate documentation for REST APIs

-> Swagger UI is used to test REST API with user interface

Assignment : https://youtu.be/ARlz2-Twm-g (watch Swagger video & practise it)

-> As of now we developed several REST API's

-> We have tested our REST APIs functionality using POSTMAN

-> We have added Swagger to generate documentation for REST API

-> We have used Swagger UI also to test REST API with User Interface

-> All our REST APIs executed in localhost (only we can access within our machine)

-> Our REST API running in the server, that server is running in local machine thats why we can access only in our local machine (It is not available for public access)

-> To provide public access for our REST API we need to deploy that into cloud platform

-> Cloud means getting resources over the web, based on our demand

-> We have several cloud platforms in the market

1) AWS

2) Azure

3) GCP

4) VM Ware

5) Heroku etc....

-> Heroku Cloud Providing 'Platform As a Service' to run our applications (PaaS)
-> Heroku Cloud Platform provided by Salesforce cloud

-> We can deploy 5 applications in Heroku for free of cost

##
App Deployment using HEROKU CLI
#######################################

1) Create Account in Heroku Cloud Platform (Free of cost)

      URL : https://dashboard.heroku.com/

2) Login into heroku, Create App in heroku cloud (5 apps we can create for free)

3) Click on App name and go to 'Deploy' section

Note: We can deploy our code into heroku app using CLI and by using Git Hub Repo

4) Download Heroku CLI Software

      URL : https://devcenter.heroku.com/articles/heroku-cli

5) Login into Heroku CLI (Open command prompt and execute below command)

```
      $ heroku login
```

Note: It will open browser to login (Login into that)

6) From cmd navigate to project folder

```
      $ cd <project-location>
```

7) Execute below commands to deploy our code into heroku cloud (heroku will provide these commands)

```
$ git init

$ heroku git:remote -a <heroku-app-name>

$ git add .

$ git commit -am "make it better"
$ git push heroku master
```

Note: With these commands, our code deployment got completed.

8) Once deployment is success, we can access our application from browser (click on Open App button)

###
How to deploy Spring Boot App in AWS Cloud
###

1) Create free tier account in AWS

2) Launch EC2 Linux Virtual Machine in AWS Cloud

3) Connect to EC2 Linux VM using MobaXterm / PuTTY

4) Upload our Spring Boot Jar file into EC2 VM

5) Install Java software in EC2 VM

6) Run the Java application in EC2 VM

7) Access the application in browser

###################################
Videos to understand AWS deployment
###################################

==> Launch EC2 VM in AWS : https://youtu.be/uI2iDk8iTps

==> Deploy Spring Boot App in AWS Cloud : https://youtu.be/cRQPgbwOWq0

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

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

-> Rest Client app and REST Api application will communicate using HTTP as a mediator

-> Every Programming language having support to develop REST Client

-> Using Spring Boot also we can develop REST Client Applications

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 response data structure

-> REST API team will send swagger documentation to REST Client side team. Using swagger documentation we need to understand the api and we need to develop REST Client logic to access REST API.

```
#####################################
RestTemplate Example to access REST API
#####################################
```

```
******************Sending HTTP  GET Request using
RestTemplate*****************


            String apiURL = "https://ashokit-sb-rest-api.herokuapp.com/";

            RestTemplate rt = new RestTemplate();

            ResponseEntity<String> forEntity = rt.getForEntity(apiURL,
String.class);

            String body = forEntity.getBody();

            System.out.println(body);
```

```
******************Sending HTTP POST Request using
RestTemplate*****************

@Service
public class BookClient {

      public void invokeBookTicket() {
            String apiUrl = "https://ashokit-book-app.herokuapp.com/book";

            Book book = new Book();
            book.setBookName("Java");
            book.setBookPrice(345.00);

            RestTemplate rt = new RestTemplate();

            ResponseEntity<String> postForEntity =
rt.postForEntity(apiUrl, book, String.class);
            System.out.println(postForEntity.getBody());
      }
}
```

```
********************Sending GET Request and binding Response JSON to
Binding Obj***********************

public void invokeGetBooksNew() {
           String apiUrl = "https://ashokit-book-
app.herokuapp.com/books";

           RestTemplate rt = new RestTemplate();
           ResponseEntity<Book[]> forEntity = rt.getForEntity(apiUrl,
Book[].class);
           Book[] body = forEntity.getBody();
           for(Book book : body) {
                System.out.println(book);
           }
     }
}

*************************************************************************
****************************
```

```
###############################
What is Synchronus & Asynchronus
#################################
```

-> Synchronus means blocking the thread until we get response

-> Asynchronus means non-blocking thread

-> RestTemplate supports only Synchronus communications

-> WebClient supports both Sync & Async communications

-> WebClient introduced in Spring 5.x version

-> WebClient is part of WebFlux starter

GET : https://ashokit-book-app.herokuapp.com/books

POST : https://ashokit-book-app.herokuapp.com/book

```
############################
HTTP Post Request with WebClient
#############################

public void invokeSaveBook() {

           Book book = new Book();
           book.setBookName("Angular");
           book.setBookPrice(450.00);

           String apiUrl = "https://ashokit-book-app.herokuapp.com/book";

           WebClient client = WebClient.create();
```

```
            String resp = client.post() // HTTP POST Request
                              .uri(apiUrl) // Endpoint URL
                              .bodyValue(book) // HTTP Request
Body Data
                              .retrieve() // Retrieve HTTP
Response Body
                              .bodyToMono(String.class) //Bind
Response to string var
                              .block(); // Make it as Sync
client

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

################################
HTTP GET Request using WebClient
#################################

```
public void invokeGetBooks() {
            String apiUrl = "https://ashokit-book-
app.herokuapp.com/books";
            WebClient client = WebClient.create();

            /*
             String body = client.get() // GET Request
                    .uri(apiUrl) // Endpoint URL
                    .retrieve() // retrieve response body
                    .bodyToMono(String.class) // bind response data to
string var
                    .block(); // make it sync
             */

            Book[] responseData = client.get()
                                        .uri(apiUrl)
                                        .retrieve()
                                        .bodyToMono(Book[].class)
                                        .block();

            for(Book b : responseData) {
                  System.out.println(b);
            }
}
```

###############################
Asynchronus call using Webclient
###############################

```
public void invokeGetBooksAsync() {
            String apiUrl = "https://ashokit-book-
app.herokuapp.com/books";
            WebClient client = WebClient.create();

                         client.get()
                                .uri(apiUrl)
                                .retrieve()
                                .bodyToMono(Book[].class)
                                .subscribe(BookClient::respHandler); //
Async Communication

            System.out.println("**************Request Sent***********");
}

public static void respHandler(Book[] books) {
            for(Book b : books) {
                  System.out.println(b);
            }
}
```

###
application.properties file vs application.yml file
##

-> When we create spring boot application by default application.properties will be created

-> We can avoid hard coded values by configuring app properties in this application.properties file

-> properties file will represent data only in key-value format

Ex:

```
server.port = 9090
spring.mvc.view.prefix = /views/
spring.mvc.view.suffix = .jsp
```

-> properties file will represent data in sequential format

-> .properties file will be supported by only java

-> For every profile we need to create a seperate properties file

******** As an alternate to .properties file we can use .yml file in spring boot ***************

-> YML stands YET ANOTHER MARKUP Language

-> YML represents data in hierarchial format

```
server:
  port: 9090
```

-> YML supports key-value, list and map values also

-> YML supported by other programming languages also (It is universal)

-> All the profiles can be configured in single YML file

############################
Working with Dynamic Properties
##############################

```
++++++++++++++++
application.yml
++++++++++++++++++++++
server:
  port: 9090
spring:
  application:
    name: sb-rest-api
messages:
  welcome: Welcome to Ashok IT..!!
  greet: Good Morning


++++++++++++++++++++++++++++++++++
@RestController
public class WelcomeRestController {

      @Value("${messages.welcome}")
      private String welcomeMsg;

      @Value("${messages.greet}")
      private String greetMsg;

      @GetMapping("/welcome")
      public String welcomeMsg() {
            return welcomeMsg;
      }

      @GetMapping("/greet")
      public String greetMsg() {
            return greetMsg;
      }
}
+++++++++++++++++++++++++++++++++++
```

-> application messages and REST ENdpoint URLS are not recommended to hardcode in java classes. Because if we change any message or any url then we have to compile and package entire application.

-> To avoid this problem we will configure messages and URLs in application.properties file or in application.yml file

-> When we change application.properties file or application.yml file we no need to compile and build entire project .

###########################
Working with App Properties
##########################

```
------------------------------application.yml----------------------------
-
spring:
  application:
    name: sb-yml-app
ashokit:
  messages:
    welcomeMsg: Welcome To Ashok IT
    greetMsg: Good Morning
    wishMsg: All the best

---------------------------AppProperties.java----------------------------
-------
@Data
@Configuration
@EnableConfigurationProperties
@ConfigurationProperties(prefix="ashokit")
public class AppProperties {

      private Map<String, String> messages = new HashMap<>();

}
-----------------------------DemoRestController.java---------------------
------
@RestController
public class DemoRestController {

      @Autowired
      private AppProperties props;

      @GetMapping("/welcome")
      public String getWelcomeMsg() {
            Map<String, String> messages = props.getMessages();
            String value = messages.get("welcomeMsg");
            return value;
      }

      @GetMapping("/greet")
      public String getGreetMsg() {
            Map<String, String> messages = props.getMessages();
            System.out.println(messages);
            String value = messages.get("greetMsg");
            return value;
      }

      @GetMapping("/wish")
      public String getWishMsg() {
            return props.getMessages().get("wishMsg");
      }
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

######################
Spring Boot Actuators
######################

-> Actuator is one of the powerful feature introduced in Spring Boot

-> Actuators are used to monitor and manage our application

-> Actuators are giving Production ready features for our boot application

+++++++++++++++++
Actuator Endpoints
+++++++++++++++++

/health : To get application health status

/info : To get application information

/beans : To get what beans loaded by our application

/mappings: To get what URL patterns available in our application

/configProps : To get configuration properties loaded by our application

/heapdump : To download heap data

/threaddump : To get threads information

/shutdown : To stop our application (This is special, it is binded to POST request)

=> To work with actuators we have to use 'spring-boot-starter-actuator' dependency

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

-> We can see actuator exposed endpoints using below URL

```
URL : http://localhost:8080/actuator
```

Note: /health is a default endpoint which we can access directly

-> We can expose other actuator endpoints using below property

```
+++++++++++++++
application.yml
+++++++++++++
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

Note: To expose endpoints using application.properties we will use below property

```
management.endpoints.web.exposure.include=*
```

+++++++++++++++++++
Working with shutdown
++++++++++++++++++++

-> IT is used to stop our application
-> We need to enable this manually

-> It is binded to http post request

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
  endpoint:
    shutdown:
      enabled: true
```

+++++++++++++++++++++++
Monlith Vs Microservices
+++++++++++++++++++++++

Application can be developed in 2 ways

1) Monolith Architecture

2) Microservices Architecture

-> If we develop all the functionalities in one single application then it is called as Monolith Application

++++++++++
Advantages
++++++++++
1) Development is easy
2) Deployment is easy
3) Performance
4) Easy Testing
5) Easy Debugging
6) KT is easy

++++++++++++++
Dis-Advantages
++++++++++++++

1) Single point of failure
2) Whole Application Re-Deployment
3) Scalability (Increasing & Decreasing resources based on demand)
4) Reliability (Strong)
5) Availability (Zero Downtime)

=> If we develop the functionalities in multiple services/apis then it is called as Microservices Architecture Based Application.

=> Every Microservice will have its own goal

++++++++++
Advantages
++++++++++

1) Loosely coupled

2) Fast Development
3) Quick Releases
4) Flexibility
5) Scalability
6) No Single Point of failure
7) Technology independence

++++++++++++++
Challenges
++++++++++++++

1) Bounded context (identifying no.of services to develop)

2) Lot of configurations

3) Visibility

4) Testing is difficult

5) Debugging

########################
Microservices Architecture
########################

-> Microservices is an architectural design pattern to develop our applications

-> There is no fixed architecture for Microservices Based Applications

-> People are customizing Microservices Architecture according to their requirement

*********** Let us see generalized architecture of Microservices ***************

1) Service Registry

2) Admin Server

3) Zipkin Server

4) Services (REST APIs)

5) FeignClient

6) API Gateway

-> ServiceRegistry is used to register all our backend services/apis
-> Service Registry will maintain services names, urls and status of each service
-> We can use EurekaServer as a service registry

Note: EurekaServer provided by Spring Cloud Netflix Library

-> AdminServer is used to monitor and manage all our backend services at one place
-> AdminServer will provide user interface to monitor and manage our services
-> Using AdminServer user interface we can access Actuator Endpoints of our services at one place

Note: AdminServer and Admin Client provided by 'Code Centric' company (we can integrate with boot)

-> ZipkinServer is used for Distributed Log Tracing
-> ZipkinServer will provide user interface to monitor application execution details
-> How many services involved and which service took more time to process the request can be monitored using Zipkin

Note: Zipkin is third party open source server (can be integrated with spring boot)

-> Backend services are nothing but REST APIs (which are also called as Microservices)
-> Backend REST APIs contains actual business logic of our application
-> One project will have multiple REST APIs in the backend
-> Each Backend api will act as client for Service Registry + Admin Server + Zipkin Server

-> With in the same application If one backend api communicating with another backend api then it is called as Interservice communication

-> FeignClient will be used to perform Interservice Communication

Note: Based on requirement our backend apis can communicate with 3 rd party apis using RestTemplate or WebClient

-> Api Gateway will act as an Entry point for our backend apis
-> It acts as mediator between endusers and backend apis
-> API Gateway contains Filters and Routing
-> Filters we can use to authenticate incoming requests
-> Routing will decide which request should go to which backend api

Note: In previous versions we have Zuul Proxy as API Gateway but now it got removed from latest version of boot

-> Spring Cloud Gateway we can use as API Gateway for our application

#####################################
Microservices Mini Project Implementation

#####################################

*********** Step- 1) Create Service Registry Application using Eureka Server ******************

1) Create Spring Boot app with below dependencies

    a) 'Spring-cloud-starter-netflix-eureka-server'
    b) web starter
    c) devtools

2) Configure @EnableEurekaServer annotation at boot start class

3) Configure below properties in application.yml file

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
```

4) Run the application and access in browser

    URL : http://localhost:8761/

************************** Step-2) Create Spring Boot Application with Admin Server ***********************************

1) Create Boot application with below dependencies

    a) web-starter
    b) devtools
    c) admin-server (code centric)

2) Configure @EnableAdminServer annotation at boot start class

3) Configure Embedded Container Port Number (we can use any port)

4) Run the application and access the application in browser

    URL : http://localhost:port/

************************** Step-3) Download & Run Zipkin Server ***

1) Download zipkin jar from below URL

    URL : https://search.maven.org/remote_content?g=io.zipkin&a=zipkin-server&v=LATEST&c=exec

2) Run the zipkin server jar file using below command

```
$   java  -jar  <zipkin-server-jar>
```

************************* Step-4) Develop REST API (WELCOME API) *********************************

1) Create boot application with below dependencies

- eureka-discovery-client
- admin-client
- zipkin client
- sleuth (It is for logging)
- web-starter
- devtools
- actuatoR

2) Configure @EnableDiscoveryClient annotation at start class (It will search and register with Eureka)

3) Create Rest Controller with required methods

4) Configure below properties in application.yml

- server port
- admin server url
- actuator endpoints
- applicaion name

```
---
server:
  port: 8081
spring:
  application:
    name: WELCOME-API
  boot:
    admin:
      client:
        url: http://localhost:1111/
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
management:
  endpoints:
    web:
      exposure:
        include: '*'
...
```

5) Run the application and check Eureka Dashboard, Admin Server Dashboard and Zipkin Dashboard

************************************Step-5) Develop REST API (GREET API)***

1) Create boot application with below dependencies

- eureka-discovery-client
- admin-client
- zipkin client
- sleuth (It is for logging)
- web-starter
- devtools
- actuator
- feign-client

2) Configure @EnableDiscoveryClient & @EnableFeignClients annotations at start class

3) Create FeginClient to access WELCOME-API

```
@FeignClient(name = "WELCOME-API")
public interface WelcomeApiClient {

      @GetMapping("/welcome")
      public String invokeWelcomeApi();

}
```

4) Create Rest Controller with required methods

5) Configure below properties in application.yml

- server port
- admin server url
- actuator endpoints
- applicaion name

```
---
server:
  port: 9091
spring:
  application:
    name: GREET-API
  boot:
    admin:
      client:
        url: http://localhost:1111/
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
management:
  endpoints:
    web:
      exposure:
        include: '*'
...
```

6) Run the application and check Eureka Dashboard, Admin Server Dashboard and Zipkin Dashboard

************************************Step-6 :: Develop API-Gateway Application***

1) Create boot application with below dependencies

- cloud-gateway
- eureka-client
- web-starter
- devtools

2) Configure @EnableDiscoveryClient annotation at boot start class

3) Configure Server Port & API Routings in application.yml file

```yaml
---
server:
  port: 3333
spring:
  application:
    name: API-GATEWAY
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
      routes:
      - id: one
        uri: lb://WELCOME-API
        predicates:
        - Path=/welcome
      - id: two
        uri: lb://GREET-API
        predicates:
        - Path=/greet
---
```

4) Run the application and Test it.

****

=> We can access Client sent request information using Filter

=> The client request information we can use to validate that request

-> Create below Filter in API Gateway (It will execute for every request)

```java
@Component
public class MyPreFilter implements GlobalFilter {

      Logger logger = LoggerFactory.getLogger(MyPreFilter.class);

      @Override
      public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
		logger.info("filter() method executed....");

		// Access request information
		ServerHttpRequest request = exchange.getRequest();

		HttpHeaders headers = request.getHeaders();

		Set<String> keySet = headers.keySet();

		keySet.forEach(key -> {
			List<String> values = headers.get(key);
			System.out.println(key + "::" + values);
		});

		return chain.filter(exchange);
	}
}
```

-> When we send request to REST API using POSTMAN, it will send POSTMAN Token in reuqest header. Using this token we can differentiate request came from other apps or from POSTMAN.

###################
Load Balancing
###################

-> If we run our application on Single Server then all requests will be sent to single server
-> Burden will be increased on the server
-> When burden got increased request processing gets delayed
-> Sometimes our server might crash due to heavy load

********** To overcome above problems we will use Load Balancing concept *****************

-> Load Balancing is the process of distributing load to multiple servers

##################################
LBR implementation in Mini Project
#################################

-> Make below changes in WelcomeApi Rest Controller

```
@RestController
public class WelcomeRestController {

	@Autowired
	private Environment env;
	@GetMapping("/welcome")
	public String welcomeMsg() {

		String port = env.getProperty("server.port");

		String msg = "Welcome to Ashok IT..!!" + " (Port :: " + port +
")";

		return msg;
	}
}
```

-> Run Welcome API with 3 instances

```
		-> Righ Click on API
		-> Run As -> Run Configurations
		-> Select Application
		-> Arguments
		-> VM Arguments (-Dserver.port=8082)
		-> Apply & Run it
```

-> Check Eureka Dashboard

##############
Circuit Breaker
##############

-> Circuit Breaker is a design pattern

-> It is used to implement fault tolerence systems

-> Fault Tolerence systems are also called as resillence systems

Requirement:

-> When m1 ( ) method failed to give response to client then m2() method should provide response to client.

```
------------------------------------------------------------------------
@RestController
public class DemoRestController {

	@GetMapping("/")
	public String m1() {
		System.out.println("********m1() method executed.....");
		String msg = "This is m1() method response";
		try {
			int i = 10 / 0;
		} catch (Exception e) {
			e.printStackTrace();
			msg = m2();
		}
		return msg;
	}
	public String m2() {
		System.out.println("********m2() method executed.....");
		String msg = "This is m2() method response";
		return msg;
	}
}
```

-> As per above program when exeception occured in 'try' block then catch block will be executed and it is calling 'm2 ( )' method.

-> When m1( ) method is failing continuosly (ex : for 5 requests) then i want to execute only m2 ( ) method directley for next 30 minutes. We can achieve this requirement by using 'Circuit Breaker'.

```
-----------------------
@SpringBootApplication
@EnableHystrix
public class Application {

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

---------------------------------------
@RestController
public class DataRestController {

	@GetMapping("/data")
	@HystrixCommand(
			fallbackMethod="getDataFromDB",
			commandProperties= {

	@HystrixProperty(name="circuitBreaker.requestVolumeThreshold",
value="5"),

	@HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds",va
lue="10000")
			}
	)
	public String getDataFromRedis() {
		System.out.println("**Redis() method called**");

		if (new Random().nextInt(10) <= 10) {
			//throw new RuntimeException("Redis Server Is Down");
		}
		// logic to access data from redis
		return "data accessed from redis (main logic) ....";
	}

	public String getDataFromDB() {
		System.out.println("**DB() method called**");
		// logic to access data from db
		return "data accessed from database (fall back logic) ....";
	}
}
----------------------------------------------------------
```

################
What is Cache ?
###############

-> Cache is a temporary storage

-> When our application wants to access same data frequently then we will use Cache memory

-> Cache will improve performance of our application by reducing database calls

Note: Database calls are always costly which will take more time to execute

-> To reduce no.of round trips between application and database we will use 'Cache'

##############
Redis Cache
###############

-> Redis is one of the distributed cache available in the market

-> Redis will store data in key-value pair

-> Multiple Applications can connect with Redis Cache at a time...

The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.

############
Redis Setup
############

-> Download Redis Software

URL : https://redis.io/download/#redis-downloads

-> Run 'redis-server.exe' file

Note: By default it runs on '6379' port number

-> Run 'Redis-cli.exe' file

-> Type 'ping' command in Redis CLI

Note: Server willl repond with 'PONG' as response

################################
Spring Boot with Redis Integration
#################################

-> Spring Boot provided starter pom to connect with Redis Server

-> Create JedisConnectionFactory bean

-> Create RedisTemplate and Inject JedisConnectionFactory into RedisTemplate

-> Using RedisTemplate get HashOperations object

-> Using HashOperations we can perform storing/retrieving/deleting operations with Redis Server

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-redis</artifactId>
			<exclusions>
				<exclusion>
					<groupId>io.lettuce</groupId>
					<artifactId>lettuce-core</artifactId>
				</exclusion>
			</exclusions>
		</dependency>
		<dependency>
			<groupId>redis.clients</groupId>
			<artifactId>jedis</artifactId>
		</dependency>
```

```
@Configuration
public class RedisConfig {

	@Bean
	public JedisConnectionFactory getJedisConnection() {
		JedisConnectionFactory factory = new JedisConnectionFactory();
		// factory.setHostName(hostName);
		// factory.setPassword(password);
		// factory.setPort(port);;
		return factory;
	}

	@Bean
	@Primary
	public RedisTemplate<String, User>
getRedisTemplate(JedisConnectionFactory factory) {
		RedisTemplate<String, User> rt = new RedisTemplate<>();
		rt.setConnectionFactory(factory);
		return rt;
	}

}
```

```
package in.ashokit.binding;

import java.io.Serializable;
import lombok.Data;

@Data
public class User implements Serializable{

      private Integer uid;
      private String name;
      private Integer age;

}

------------------------------------------
@RestController
public class UserRestController {

      private HashOperations<String, Integer, User> hashOps;

      public UserRestController(RedisTemplate<String, User>
redisTemplate) {
            hashOps = redisTemplate.opsForHash();
      }

      @PostMapping("/user")
      public String storeData(@RequestBody User user) {
            hashOps.put("PERSONS", user.getUid(), user);
            return "success";
      }

      @GetMapping("/user/{uid}")
      public User getData(@PathVariable Integer uid) {
            User value = (User) hashOps.get("PERSONS", uid);
            return value;
      }

      @GetMapping("/users")
      public List<User> getAllUsers(){
            return hashOps.values("PERSONS");
      }

      @DeleteMapping("/user/{uid}")
      public String deleteUser(@PathVariable Integer uid) {
            hashOps.delete("PERSONS", uid);
            return "User Deleted";
      }
}
```

##############
Apache Kafka
#############

-> Apache Kafka is an open source distributed streaming platform

-> Apache Kafka is used to process real time data

-> We will use Apache Kafa as a message broker for our applications

-> Kafka works based on Publisher and Subscriber Model

Note: Kafka will act as a mediator / broker between Publisher and Subscriber

-> The application which is publishing message to kafka is called Publisher

-> The application which is subscribing message from kafka is called Subscriber

Note: Using apache kafka we can develop Event Driven Microservices

+++

==> Kafka - Workshop Video : https://youtu.be/VInk1_9vvCY

==> Kafka Notes + Example : https://github.com/ashokitschool/ashokit_weekend_workshops.git

+++

##############
Config Server
##############

-> As part of our application development we will use several configuration properties

Ex:

a) data source properties
b) actuator properties
c) security properties
d) smtp properties
e) kafka properties
f) application messages etc..

-> As of now we configured those configuration properties in application.properties / application.yml file

-> application.properties / application.yml file will be available with in the project

-> When we package our boot application for depeloyment our configuration properites will be part of that packaged file

Note: If we want to change any configuration properties then we have to package our application again and we have to re-deploy our application (This is not recommended).

-> To avoid this problem we will keep configuration properties outside of the project.

****************** Config server is used to externalize application configuration properties ********************************

-> Using Config Server we can load Configuration Properties from outside of the project

-> When we want to change any configuration properties we no need to re-package and re-deploy our application

-> Using Config Server we can de-couple our application and configuration properties

****

1) Create Git Hub Repository and keep configuration properties in git hub repo

Note: We need use application name for configuration properties/yml file name

Ex:

welcome.yml
welcome-dev.yml
welcome-prod.yml

admin.yml
admin-dev.yml
admin-prod.yml

reports.yml
reports-dev.yml
reports-prod.yml

Git Repo URL :
https://github.com/ashokitschool/configuration_properties.git

#####################
Config Server Project
######################

1) Create Boot application with below dependencies

   a) config-server
   b) actuator

2) Write @EnableConfigServer annotation at boot start class

3) Configure below properties in application.yml file

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/ashokitschool/configuration_properties
          clone-on-start: true
management:
  security:
    enabled: false
```

##
Microservice To Load Config Properties using Config Server (Config Client App)
##

1) Create Boot application with below dependencies

    a) config-client
    b) web-starter
    c) cloud-bootstrap

2) Configure application name, application port, config-server-url, profile

a) bootstrap.yml (app-name & config-server-url)

```yaml
spring:
  application:
    name: welcome
  cloud:
    config:
      uri: http://localhost:8080
```

b) application.yml (server port)

```yaml
server:
  port: 9090
```

3) Create Rest Controller with required methods

```java
@RestController
@RefreshScope
public class WelcomeRestController {

      @Value("${msg:Config Server Not Working}")
      private String msg;

      @GetMapping("/")
      public String getWelcomeMsg() {
            return msg;
      }
}
```

4) Run the application and test it.

####################
Mono & Flux Objects

####################

-> Mono means single object

-> Flux means stream of objects

-> Create spring boot application with below 3 dependencies

1) web-starter
2) webflux
3) lombok

```
--------------------------------
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CustomerEvent {

      private String name;
      private Date createDate;

}
---------------------------------
@RestController
public class CustomerRestController {

      @GetMapping(value = "/event", produces = "application/json")
      public ResponseEntity<Mono<CustomerEvent>> getEvent() {
            CustomerEvent event = new CustomerEvent("Ashok", new Date());
            Mono<CustomerEvent> customerMono = Mono.just(event);
            return new ResponseEntity<Mono<CustomerEvent>>(customerMono,
HttpStatus.OK);
      }

      @GetMapping(value = "/events", produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
      public ResponseEntity<Flux<CustomerEvent>> getEvents() {

            // creating binding object with data
            CustomerEvent event = new CustomerEvent("Ashok", new Date());

            // creating stream for binding object
            Stream<CustomerEvent> customerStream = Stream.generate(() ->
event);

            // create flux object using stream
            Flux<CustomerEvent> cflux = Flux.fromStream(customerStream);

            // setting response interval
            Flux<Long> intervalFlux =
Flux.interval(Duration.ofSeconds(5));

            // combine interval flux and customer event flux
            Flux<Tuple2<Long, CustomerEvent>> zip = Flux.zip(intervalFlux,
cflux);

            // Getting Tuple value as T2
            Flux<CustomerEvent> fluxMap = zip.map(Tuple2::getT2);
            //sending response
            return new ResponseEntity<>(fluxMap, HttpStatus.OK);
      }
}
```

#############################
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 one CODE i.e exception code

-> In Spring web mvc we can handle exceptions in 2 ways

1) Controller Based Exception Handling

                 - Exception Handlers applicable for only particular controller

2) Global Exception Handling

                 - Exception Handlers applicable for all the classes in the project

```
--------------------------------
@Data
public class ExceptionInfo {

      private String msg;
      private String code;

}
------------------------------------------
@RestController
public class DemoRestController {

      private Logger logger = 
LoggerFactory.getLogger(DemoRestController.class);

      @GetMapping("/")
      public String doAction() {
            String msg = "Action in progress";
            try {
                  int i = 10 / 0;
            } catch (Exception e) {
                  logger.error("Exception Occured ::" + e, e);
                  throw new ArithmeticException(e.getMessage());
            }
            return msg;
      }

      @ExceptionHandler(value=ArithmeticException.class)
      public ResponseEntity<ExceptionInfo> handleAE(ArithmeticException 
ae) {
            ExceptionInfo exception = new ExceptionInfo();

            exception.setMsg(ae.getMessage());
            exception.setCode("AIT0004");

            return new ResponseEntity<>(exception, 
HttpStatus.INTERNAL_SERVER_ERROR);
      }
}
--------------------------------------------------------------------------
-----------------------------------------
```

##############
Spring Security
###############

-> To implement security for our applications Spring provided 'security' module

-> To use Spring Security in our project 'spring-security-starter' we need to add in pom.xml file

```
            <dependency>
                  <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
```

-> By default it will secure all the endpoints of our application

```
        Default uname : user
        Default Pwd: Will be printed on the console
```

-> To override default credentials we can configure credentails in application.properties file

```
spring.security.user.name=admin
spring.security.user.password=admin@123
```

-> Create Rest Controller class with required method

#####################################
Rest Client To access Secured REST API
#######################################

```
@Service
public class WelcomeService {

      private String apiUrl = "http://localhost:8080";

      public void invokeWelcomeApi() {

            RestTemplate rt = new RestTemplate();

            HttpHeaders headers = new HttpHeaders();
            headers.setBasicAuth("admin", "admin@123");

            HttpEntity<String> reqEntity = new HttpEntity<>(headers);

            ResponseEntity<String> responseEntity = rt.exchange(apiUrl,
HttpMethod.GET, reqEntity, String.class);

            String body = responseEntity.getBody();

            System.out.println(body);
      }


      public void invokeWelcome() {

            WebClient webClient = WebClient.create();

            String block = webClient.get()
                                          .uri(apiUrl)
                                          .headers(headers ->
headers.setBasicAuth("admin", "admin@123"))
                                          .retrieve()
                                          .bodyToMono(String.class)
                                          .block();

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

*************************** Create Spring Boot Application with below starters ********************

a) web-starter
b) data-jpa
c) h2
d) project lombok
e) devtools

************************** Create Entity class **

```
@Data
@Entity
@Table(name = "BOOK_DTLS")
public class Book {

      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      @Column(name = "BOOK_ID")
      private Integer bookId;

      @Column(name = "BOOK_NAME")
      private String bookName;

      @Column(name = "BOOK_PRICE")
      private Double bookPrice;

}
```

******************************* Create Repository interface **

```
public interface BookRepository extends JpaRepository<Book, Serializable>{

}
```

*********************************** Create Service interface and impl class *****************************

```
public interface BookService {

      public String upsertBook(Book book);

      public List<Book> getAllBooks();

}

@Service
public class BookServiceImpl implements BookService {

      private BookRepository repository;

      public BookServiceImpl(BookRepository repository) {
            this.repository = repository;
      }
```

```
	@Override
	public String upsertBook(Book book) {
		repository.save(book);
		return "Record Inserted";
	}

	@Override
	public List<Book> getAllBooks() {
		return repository.findAll();
	}
}
```

** Create Rest Controller **************************************

```
@RestController
@CrossOrigin
public class BookRestController {

	@Autowired
	private BookService service;

	@PostMapping("/book")
	public ResponseEntity<String> addBook(@RequestBody Book book) {
		String msg = service.upsertBook(book);
		return new ResponseEntity<>(msg, HttpStatus.CREATED);
	}

	@GetMapping("/books")
	public ResponseEntity<List<Book>> getAllBooks() {
		List<Book> allBooks = service.getAllBooks();
		return new ResponseEntity<>(allBooks, HttpStatus.OK);
	}
}
```

******************************** Configure below properties in application.properties file ************************

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.hibernate.ddl-auto=update

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

************************************** Run the boot application and insert the data using POST Request **************

******************** Create Angular application ************************

```
	$ ng new bookapp
```

******************** Create Book class to represent json response in object format *****************

```
      $ ng generate class Book

export class Book {

    bookId:number;
    bookName:string;
    bookPrice:number;

     constructor(a:number,b:string,c:number){
        this.bookId = a;
        this.bookName = b;
        this.bookPrice = c;
    }
}
```

****************** Import HttpClientModule & FormsModule in AppModule***************************

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule, FormsModule, HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

****************************   Write REST Call logic in AppComponent *****************************

```
import { HttpClient } from '@angular/common/http';
import { Component, Inject } from '@angular/core';
import { Book } from './book';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  msg:string="";
  book:Book = new Book(1,"Spring", 200);
  books:Book[] = [];

  constructor(@Inject(HttpClient)private http:HttpClient){}

  getData(){
    this.http.get<Book[]>("http://localhost:8080/books", {responseType :
'json'})
    .subscribe(data => {
      this.books = data;
    });
  }

  onInsertClick(){
    this.http.post("http://localhost:8080/book", this.book,
{responseType:"text"})
    .subscribe(data => {
    this.msg = data;
  });
 }

}
```

*** Write presentation logic in template *****************************

```
<div>
  <h3>Angular UI + Boot REST API</h3>
  <form>
  Book ID : <input type="text" name="bookId"
[(ngModel)]="book.bookId"/><br/>
  Book Name : <input type="text" name="bookName"
[(ngModel)]="book.bookName"/><br/>
  Book Price : <input type="text" name="bookPrice"
[(ngModel)]="book.bookPrice"/><br/>
  <input type="submit" value="Save Book" (click)="onInsertClick()"/><br/>
  {{msg}}
  </form>
</div>
<div>
  <h3>Book Details</h3>
      <input type="button" value="Get Data" (click)="getData()"/>

      <table border="1">
          <tr>
              <th>Book Id</th>
              <th>Book Name</th>
              <th>Book Price</th>
          </tr>
          <tr *ngFor="let book of books">
              <td>{{book.bookId}}</td>
              <td>{{book.bookName}}</td>
              <td>{{book.bookPrice}}</td>
          </tr>
      </table>
  </div>
```

Run the Angular Application *************

What is Unit testing ?
++++++++++++++++++

-> It is the process of testing unit amount of work

-> When we implement code, we need to test weather that code is working or not

-> With the help of unit testing we can identify issues in the code

-> To perform Unit testing we will use Junit

-> Junit is an open source & free framework to perform unit testing for java applications

############
Mocking
#############

-> Mocking is the process of creating substitute object for the real object

-> Using Mock Objects we can perform isolated unit testing

```
*************************************************************************
@Service
public class WelcomeService {

      public String getMsg() {
            String msg = "Good Morning";

            return msg;
      }
}
------------------------------------------------------------
@RestController
public class WelcomeRestController {

      @Autowired
      private WelcomeService service;

      @GetMapping("/welcome")
      public String welcomeMsg() {
            String msg = service.getMsg();
            return msg;
      }
}
--------------------------------------------------------------
@WebMvcTest(value = WelcomeRestController.class)
public class WelcomeRestControllerTest {

      @MockBean
      private WelcomeService service;

      @Autowired
      private MockMvc mockMvc;

      @Test
      public void welcomeMsgTest() throws Exception {

            // defining mock obj behaviour
            when(service.getMsg()).thenReturn("Welcome to Ashok IT");
            // preparing request
            MockHttpServletRequestBuilder reqBuilder =
MockMvcRequestBuilders.get("/welcome");

            // sending request
            MvcResult mvcResult = mockMvc.perform(reqBuilder).andReturn();

            // get the response
            MockHttpServletResponse response = mvcResult.getResponse();

            // validate response status code
            int status = response.getStatus();
            assertEquals(200, status);

      }

}
```

-> Angular is a client side framework developed by Google company

-> Angular framework is developed based on TypeScript

-> Angular is mainley used for Single Page Applications

-> Angular supports multiple devices (Mobiles & Desktops)

-> Angular supports multiple browsers

-> Angular is free and open source

-> Angular JS and Angular both are not same

-> From Angular 2 version onwards it is called as Angular Framework

Note: The current version of Angular is 14

Angular Building Blocks
----

1) Component
2) Metadata
3) Template
4) Data Binding
5) Module
6) Service
7) Dependency Injection
8) Directive
9) Pipes

-> Angular application is collection of components. In components we will write logic to send data to template and capture data from template. Components are TypeScript classes.

-> Metadata nothing data about the data. It provides information about components.

-> Template is a view where we will write our presentation logic. In Angular application template is a HTML file. Every Component contains its own Template.

-> Data Binding is the process of binding data between component property and view element in template file.

-> Module is a collection of components directives and pipes

-> Service means it contains re-usable business logic. Service classes we will inject in Components using Depdency Injection.

-> Dependency Injection is the process of injecting dependent object into target object. In Angular applications services will be injected into components using DI.

-> Directives are used to manipulate DOM elements.

-> Pipes are used to transform the data before displaying

Environment Setup For Angular Applications

1) Install Node JS
2) Install TypeScript
3) Install Angular CLI
4) Install Visual Studio Code IDE

-> Angular 2+ framework is available as a collection of packages, those packages are available in "Node". To use those packages "npm" (Node Package Manager) is must and should..

      URL : https://nodejs.org/en/

-> After installing node software, open cmd and type node -v. It should display node version number then installation is successfull.

-> Angular framework itself is developed based on TypeScript. In Angular applications we will write code using TypeScript only.

-> We can install Typescript using Node Package Manager (npm). Open command prompt and execute below command to install TS.

```
$ npm install -g typescript
```

-> After TypeScript installation got completed we can verify version number using below command in cmd. If it displays version number then installtion is successfull.

```
$ tsc -v
```

-> Install Angular CLI software using below command in TypeScript.

```
$ npm install @angular/cli -g
```

-> Check angular installation using below command

```
$ ng v
```

-> Download and install VS code IDE

    URL : https://code.visualstudio.com/download

Note: We are done with angular setup... lets start building angular applications

SPRING BOOT INTERVIEW QUESTIONS

Q1. What is Spring Boot?

Ans: Spring Boot is used to create stand-alone, production ready Spring based applications that can just run.

Spring boot internally uses Spring framework which helps in making the application development easy and faster. It is built on top of Spring framework.

It helps in developing the microservice based application.

Q2. Advantages of Spring Boot?

Ans: The main advantages of Spring Boot are:

Auto Configuration – It helps in automatically configuring the application based on the dependencies added in the classpath. When using spring, to configure a datasource , a lot of configuration is need to configure entity manager ,transaction manager,etc. But Spring boot reduces to minimal configuration and uses the existing configuration.

Starter POMS – It consists of multiple starter POMS which are mainly used to reduce maven configuration. It helps in maintaining the POM more easily as number of dependencies are reduced.

Actuators – This helps in providing the production ready features of the application such as health check, metrics, classes loaded by the application,etc.

Rapid Application Development – Spring Boot provides the infrastructure support which is required for our application and hence the application can be developed in the quickest manner.

Embedded Servers – It comes with embedded servers such as Tomcat, Jetty etc without the need to set up an external server.

Embedded Database Integration – It also supports integration with the embedded database such as H2 database.

Q3. What is the internal working of @SpringBootApplication annotation?

Ans: It is a combination of three annotations @ComponentScan, @EnableAutoConfiguration and @Configuration.

@Configuration: It is a class level annotation which indicates that a class can be used by the Spring IOC container as a source of bean definition. The method annotated with @Bean annotation will return an object that will be registered as a Spring Bean in IOC.

@ComponentScan: It is used to scan the packages and all of its sub-packages which registers the classes as spring bean in the IOC container.

@EnableAutoConfiguration: This annotation tells how Spring should configure based on the jars in the classpath. For eg , if H2 database jars are added in classpath , it will create datasource connection with H2 database.

Q4. What is Spring Initializer.

Ans: Spring initializer is a web - based tool which is used to create a project structure for spring based applications. It does not generate any source code but helps in creating a project structure by providing maven or gradle build tool for building the application.

Q5. What is the default port number of tomcat in Spring Boot? Is it possible to change the port number?

Ans: The default port number of tomcat is 8080, yet it is possible to override it using the property server.port = port_number in application.properties or application.yml.

Q6. What are the spring boot starters?

Ans: Spring Boot provides multiple starter projects which are needed to develop different types of web application. For e.g., if we add spring-boot-starter-web as a dependency in pom.xml, we can develop mvc applications. All the dependency jars will be added to the classpath which are required for the application development using MVC.

Some of the starter POMS are:

- *Spring-boot-starter – It helps in developing stand-alone applications.*
- *Spring-boot-starter-web - It helps in designing web based and distributed applications.*
- *Spring-boot-starter-data-jpa – used in designing the persistence layer.*

It reduces the code for maven configuration.

Q7. Explain the need of dev-tools dependency.

Ans: The main aim of adding dev-tools dependency is to improve the development time. When this dependency is included in the project, it automatically restarts the server when there are any modifications made to the source code which helps in reducing the effort of a developer to manually build and restart the server.

Q8. Difference between Spring and Spring Boot.

Ans: Spring – It is an opensource J2EE framework which helps in developing web based and enterprise applications easily. Its main feature is dependency injection by which we can achieve loosely coupling while developing the application. In order to develop a web application, developer needs to write a lot of code for configuring the dispatcher servlet in web.xml, configuring the database,etc. This can be avoided while using Spring Boot.It does not support embedded servers and embedded database integration.

Spring Boot – Spring Boot is built on top of Spring framework which provides flexibility to design applications in a rapid and easier approach. It provides auto configuration feature through which it reduces the developer's effort to write large xml configuration. It provides support for embedded server without the need to install it explicitly.

Q9. How to create a spring boot project using Spring Initializer.

Ans: Spring initializer is a web-based tool developed by Pivotal. With the use of it, we can easily create the project structure needed to develop Spring based application.

- *Go to the official Spring Initialize website:* ***https://start.spring.io***

- *Select the project details such as language, spring boot version, build tool which is needed for application development as:*
    - *Language: Java*
    - *Java Version: 1.8*
    - *Spring Boot: 2.1.4*
- *Add the required dependencies and click on Generate project. It shall the download the project in your system.*
- *Import the zip file into the eclipse.*

Q10. What is component scanning?

Ans: Component scanning is the process of identifying the Spring beans in the packages and its sub packages. In a spring boot application, the packages which contains the SpringBootApplication class is called as base package and will be scanned implicitly.

@ComponentScan is used to scan the packages and detect the spring beans which will be managed by the IOC container.

If we have more than one base package, then we need to scan the base package using @ComponentScan in Spring start class.

Syntax:

@ComponentScan(basePackages = "in.ashokit.service")

Q11. What is a Spring Bean?

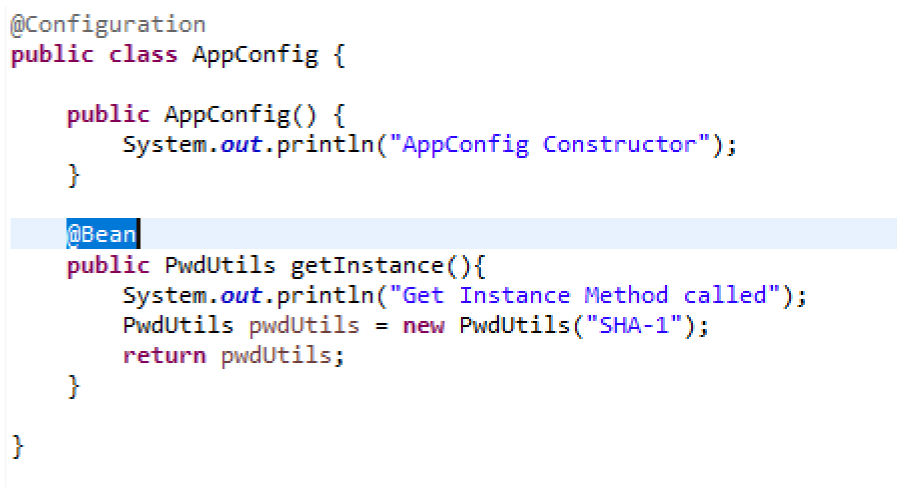
Ans: A java class which is managed by the IOC container is called as Spring Bean.The life cycle of the spring bean are taken care by the IOC container.

A spring bean can be represented by using the below annotations.

- *@Component*
- *@Service*
- *@Repository*
- *@Configuration*
- *@Bean*
- *@Controller*
- *@RestController*

Q12. What is the use of @Configuration annotation?

Ans: A class which is used to provide few configurations such as Swagger configuration, Kafka configuration,etc can be represented using @Configuration annotation. This class contains Bean methods to customize the object creation and returns the object which can be respresented as a Spring Bean by the IOC container.



```java
@Configuration
public class AppConfig {

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

    @Bean
    public PwdUtils getInstance(){
        System.out.println("Get Instance Method called");
        PwdUtils pwdUtils = new PwdUtils("SHA-1");
        return pwdUtils;
    }

}
```

Q13. What is Auto-wiring?

Ans: Autowiring is the process of injecting one class object into another class. It cannot be implied on primitive types and String type.

Autowiring can be done in 3 ways:

- *Constructor Injection*
- *Setter Injection*
- *Field Injection*

Q14. What is a Runner and its use?

Ans: Runner classes are used to execute the piece of code as soon as the application starts. The code inside the runner classes will execute once on bootstrap of the application. There are mainly used to setup a data source, load the data into cache, etc. These runners will be called from SpringApplication.run() method.

There are two types of Runner Interfaces.

- *ApplicationRunner*
- *CommandLineRunner*

Q15. What is ApplicationRunner in SpringBoot?

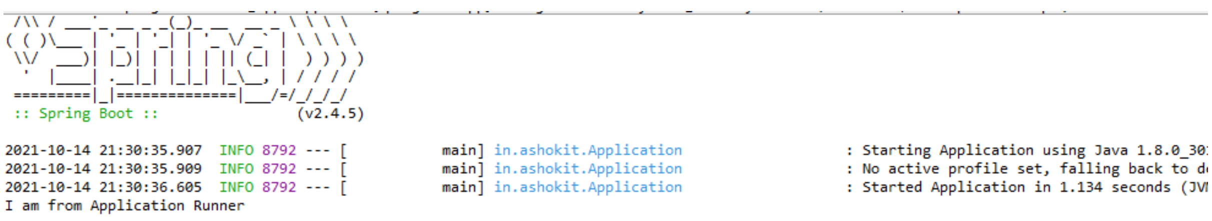
Ans: Application Runner is a functional interface which contains only one abstract method run().

When there is a need to execute some piece of code during the bootstrap of the spring boot application, then we need to write a Runner class to override the run method and provide the implementation.

Example:

```java
@Component
public class AppRunner  implements ApplicationRunner{

    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("I am from Application Runner");

    }

}
```

Output:



```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.5)

2021-10-14 21:30:35.907  INFO 8792 --- [           main] in.ashokit.Application                   : Starting Application using Java 1.8.0_30
2021-10-14 21:30:35.909  INFO 8792 --- [           main] in.ashokit.Application                   : No active profile set, falling back to d
2021-10-14 21:30:36.605  INFO 8792 --- [           main] in.ashokit.Application                   : Started Application in 1.134 seconds (JV
I am from Application Runner
```

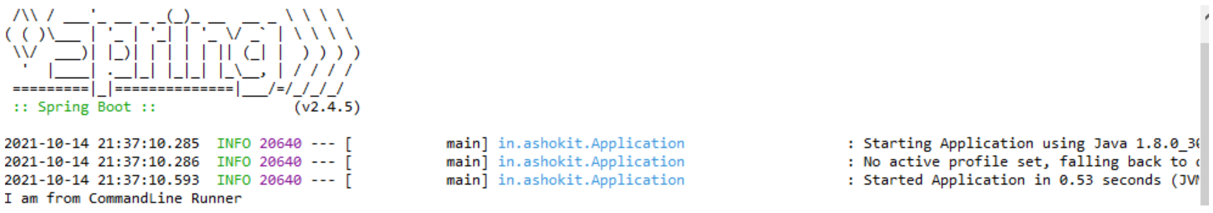
Q16. What is CommandLineRunner in SpringBoot?

Ans: CommandLineRunner is similar to the ApplicationRunner interface which is also used to execute the logic only once during application startup. The only difference between CommandLineRunner and ApplicationRunner is that ApplicationRunner accepts the arguments in the form of ApplicationArguments where as CommandLineRunner accepts in String[] array.

Example: A class implementing CommandLineRunner interface.

```java
import org.springframework.boot.CommandLineRunner;

@Component
public class CmdRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("I am from CommandLine Runner");

    }

}
```

Output :

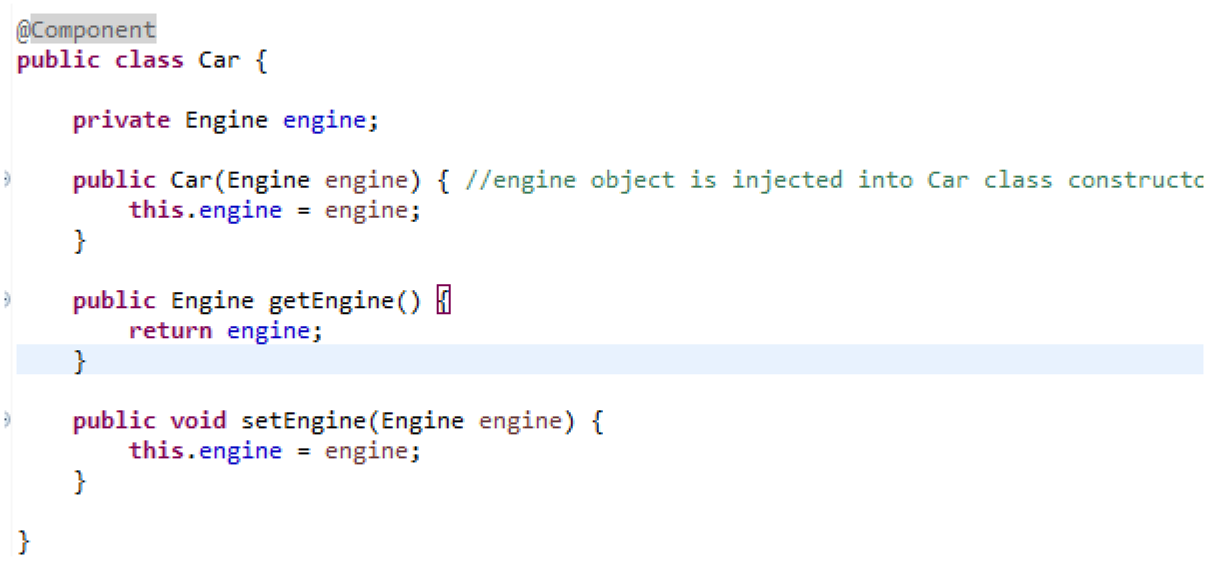


```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.5)

2021-10-14 21:37:10.285  INFO 20640 --- [           main] in.ashokit.Application                   : Starting Application using Java 1.8.0_3
2021-10-14 21:37:10.286  INFO 20640 --- [           main] in.ashokit.Application                   : No active profile set, falling back to
2021-10-14 21:37:10.593  INFO 20640 --- [           main] in.ashokit.Application                   : Started Application in 0.53 seconds (JV
I am from CommandLine Runner
```

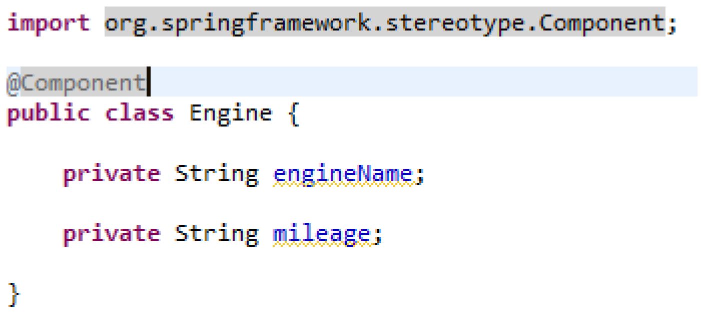
Q17. Explain Constructor Injection.

Ans: Constructor Injection is the process of injecting the dependent bean object into the target bean using the target class construction.

E.g. : If a class Car is dependent on the Engine object which is needed for Car to run, in this case Engine object will be created first and dependency will be injected into Car class. If the dependency is achieved using the target class constructor, it is referred to as Constructor injection.



```java
@Component
public class Car {

    private Engine engine;

    public Car(Engine engine) { //engine object is injected into Car class constructc
        this.engine = engine;
    }

    public Engine getEngine() {
        return engine;
    }

    public void setEngine(Engine engine) {
        this.engine = engine;
    }

}
```



```java
import org.springframework.stereotype.Component;

@Component
public class Engine {

    private String engineName;

    private String mileage;

}
```

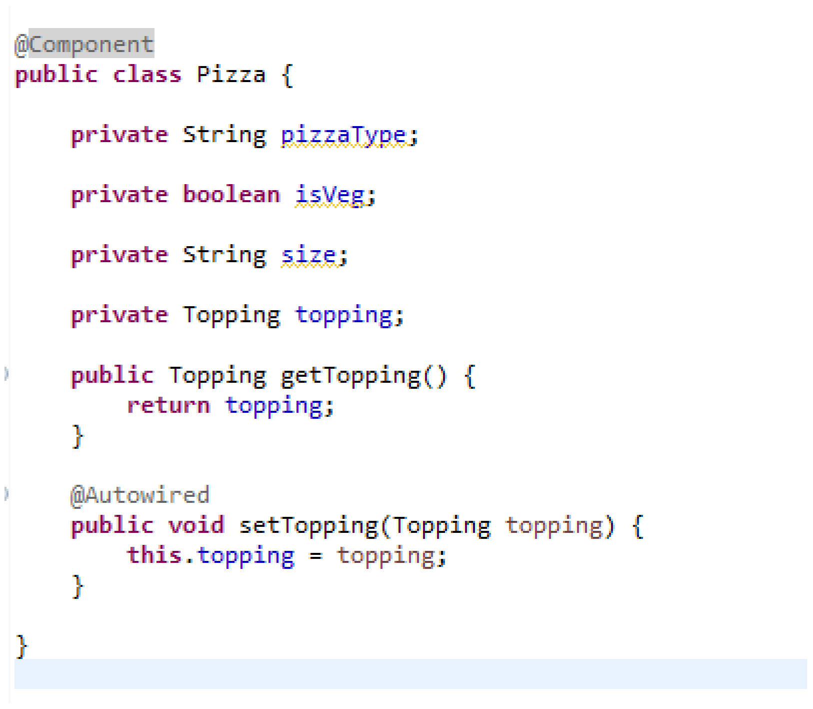
It is not mandatory to give @Autowired annotation if there is only one constructor.

Q18.Explain Setter Injection.

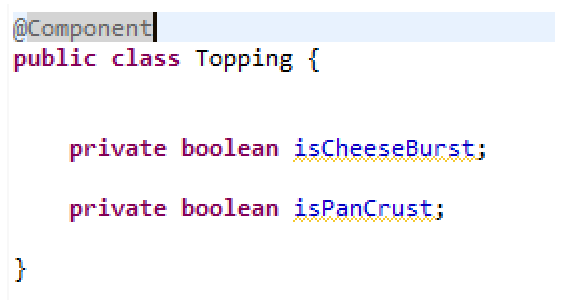
Ans: Setter injection is another mechanism to perform dependency injection. In this approach, the dependent object is injected into target class using target class setter methods. Setter injection can override the constructor injection.

@Autowired annotation is used on the setter methods.

Eg:



```
@Component
public class Pizza {

    private String pizzaType;

    private boolean isVeg;

    private String size;

    private Topping topping;

    public Topping getTopping() {
        return topping;
    }

    @Autowired
    public void setTopping(Topping topping) {
        this.topping = topping;
    }

}
```



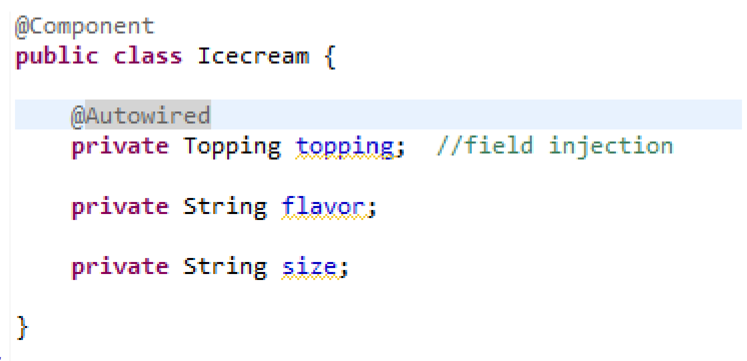
```
@Component
public class Topping {


    private boolean isCheeseBurst;

    private boolean isPanCrust;

}
```

In setter injection, target class object should be created first followed by dependent object.

Q19.Explain Field Injection.

Ans: Field injection is a mechanism where the dependent object is injected into the target object using target class variable directly with the use of @Autowired annotation. It internally uses Reflection API to perform field injection

Eg:



```
@Component
public class Icecream {

    @Autowired
    private Topping topping;   //field injection

    private String flavor;

    private String size;

}
```

:

Q20. Can you override the default web server in Spring Boot.

Ans: By default, Spring Boot provides Tomcat as the embedded server. This can be changed, as we can configure Jetty, Netty as embedded servers in Spring boot. This can be done in a convenient way by adding the starter dependencies in the maven pom.xml.

Example: Adding Jetty as dependency to pom.xml

```
<dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-jetty</artifactId>

</dependency>
```

SPRING - JPA

Q21. What is Spring Data JPA?

Ans: Spring data JPA is used for designing the persistence layer of the web application. It is used for managing the relational data in a java application. It acts as an intermediate between the java object and relational database.

Spring data JPA is mainly built on top of JDBC API and it helps in reducing the boilerplate code.

Spring boot provides a starter-POM “spring-boot-starter-data-jpa” which is used to design the DAO layer. All the required jars are added to the classpath after adding the starter pom in dependency management configuration file.

It provides predefined interfaces which has methods to perform CRUD operations.

Q22. Explain features of Spring Data JPA?

Ans: The main advantages of using Spring Data are:

No-code repositories: *Spring data provides predefined repository interfaces which should be extended to create a repository for the entity class. It has the built-in methods to perform the CRUD operation.*

Reduces Boilerplate code: *It reduces a lot of boiler plate such as creating a connection object, creating a statement and executing the query,closing the resources,etc.*

Spring data provides predefined methods which are already implemented in the Repository interfaces, and by just calling those methods, we can perform CRUD operations.

Generation of the queries: *Another feature is queries are automatically generated based on the method names.*

Eg: If there is a method in EmployeeRepository as:

```
public List<Employee> findByName(String empName);
```

Spring data jpa will create a query as below :

select e.empid,e.empname,e.esalary from employee e where e.name = ? ;

Pagination and Sorting support: *It supports pagination and sorting using predefined interface PagingAndSortingRepository.*

Q23. How to create a custom Repository class in Spring JPA?

Ans: We can create custom repository by extending one of the interfaces as below:

- *CrudRepository*
- *JpaRepository*
- *PagingAndSortingRepository*

```
import java.io.Serializable;

import org.springframework.data.jpa.repository.JpaRepository;

import in.ashokit.entities.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Serializable> {

}
```

Q24. Difference between CRUDRepository and JPARepository.

Ans: CrudRepository interface provides method to perform only crud operations. It allows to create , read, update and delete records without creating own methods.

JPARepository extends PagingAndSortingRepository which provides methods to retrieve records using pagination and also to sort the records.

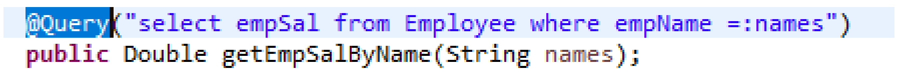
PagingAndSortingRepository extends CrudRepository which allows to do CRUD operations.

Q25. Write a custom query in Spring JPA?

Ans: A custom query can be written using @Query annotation in the Repository interface. Using this annotation, we can write HQL queries and native SQL queries.

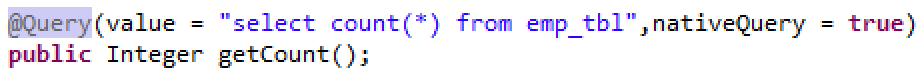
HQL queries can be written as below to fetch Emp Salary based on name.

Eg :



```
@Query("select empSal from Employee where empName =:names")
public Double getEmpSalByName(String names);
```

Example for a native sql query is,



```
@Query(value = "select count(*) from emp_tbl",nativeQuery = true)
public Integer getCount();
```

Q26. What is the purpose of save () method in CrudRepository.

Ans: An entity can be saved into the database using save () method of CrudRepository. It will persist or merge the entity by using JPA Entity Manager. If the primary id is empty, it will call entityManager.persist(…) method, else it will merge the existing record by making a call to entityManager.merge(…) method.

Q27. Difference between findById() and getOne().

Ans: The findById() method is available in CrudRepository while getOne() is available in JpaRepository.

The findById() returns null if record does not exist while the getOne() will throw an exception called EntityNotFoundException.

getOne() is a lazy operation where it will return a proxy without even hitting the database.

findById() – will retrieve the row record by directly hitting the database.

Q28. Use of @Temporal annotation.

Ans: Prior to Java 8, @Temporal annotation is mainly used to convert the date and time values of an object to the compatible database type.

Generally, when we declare a Date field in the class and try to store it.It will store as TIMESTAMP in the database.

Eg:

```
@Temporal

private Date DOJ;
```

Above code will store value looks like 08-07-17 04:33:35.52000000 PM.

We can use TemporalType to DATE if the requirement is to store only date.

```
@Temporal (TemporalType.DATE)

private Date DOJ;
```

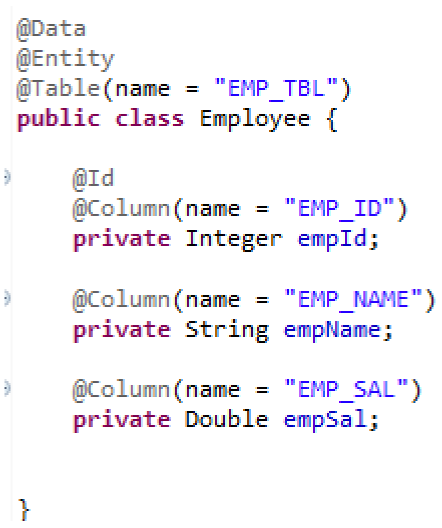
But after Java 8, there is no need to use @Temporal due to the introduction of LocalDate and LocalTime Api.

Q29. Write a query method for sorting in spring data jpa.

Ans: Spring data JPA provides two ways to sort the records in ascending and descending manner.

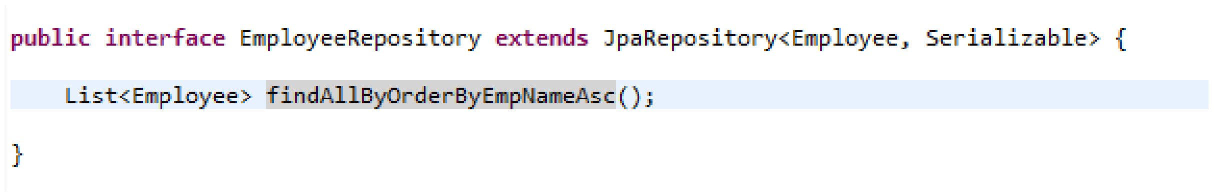
Approach 1: Using OrderBy method

E.g.: If there is an Entity class as:



```java
@Data
@Entity
@Table(name = "EMP_TBL")
public class Employee {

    @Id
    @Column(name = "EMP_ID")
    private Integer empId;

    @Column(name = "EMP_NAME")
    private String empName;

    @Column(name = "EMP_SAL")
    private Double empSal;


}
```

And the requirement is to fetch all the employees and sort by name in ascending order, then write a custom method as:



```java
public interface EmployeeRepository extends JpaRepository<Employee, Serializable> {

    List<Employee> findAllByOrderByEmpNameAsc();

}
```

Approach 2: Using Sort.by method

To sort the same above requirement, we can write the method as below

List<Employee> employees = empRepo.findAll(Sort.by(Sort.Direction.ASC, "empName"));

Q30. Explain @Transactional annotation in Spring.

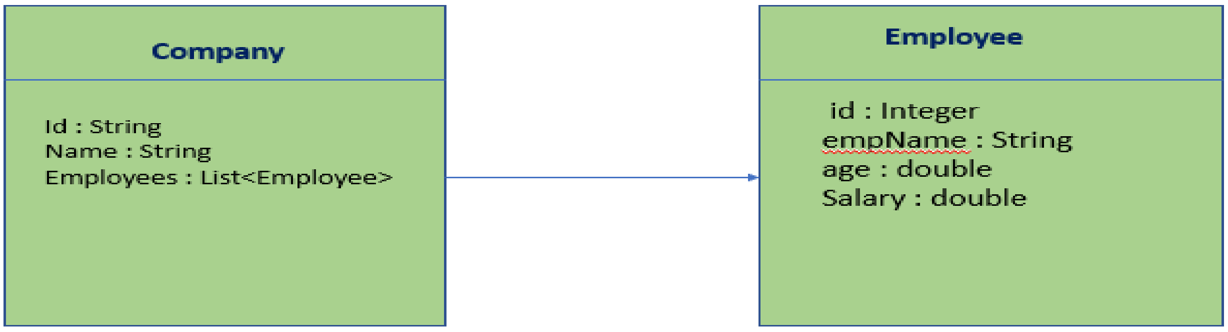
Ans: A database transaction is a sequence of statements/actions which are treated as a single unit of work. These operations should execute completely without any exception or should show no changes at all. The method on which the @Transactional annotation is declared, should execute the statements sequentially and if any error occurs, the transaction should be rolled back to its previous state. If there is no error, all the operations need to be committed to the database. By using @Transactional, we can comply with ACID principles.

E.g.: If in a transaction, we are saving entity1, entity2 and entity3 and if any exception occurs while saving entity3, then as enitiy1 and entity2 comes in same transaction so entity1 and entity2 should be rolledback with entity3.

A transaction is mainly implied on non-select operations (INSERT/UPDATE/DELETE).

Q31. What is the difference between FetchType.Eager and FetchType.Lazy?

Ans: If there exists a relationship between two entity classes, in this case for eg: Company Entity and an Employee entity as shown in the diagram.



In the above diagram, there exists a one-to-many relationship between Company Entity and Employee Entity.

When you are trying to load the company details, it will load the id, name columns, etc. But it will not load the employee details. Employee details can be loaded in two ways.

FetchType.LAZY – It will not load the Employee details while firing the query to get Company Data. This is called as lazy loading where in the employee details will be loaded on demand.

FetchType.EAGER – This will load all the employee details while loading the Company data.

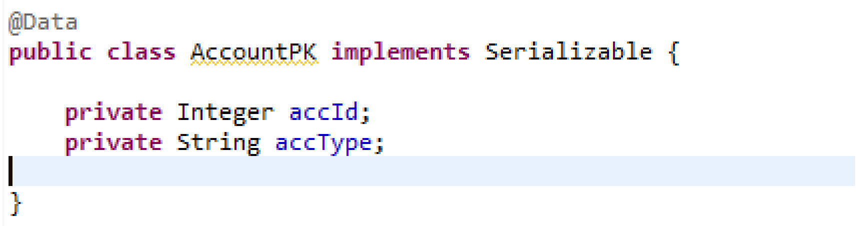
Q32. Use of @Id annotation.

Ans: @Id annotation is used on a field of an Entity class to mark the property/column as a primary key in the database. It is used along with @GeneratedValue which is used to generate the unique primary keys.

Q33. How will you create a composite primary key in Spring JPA.

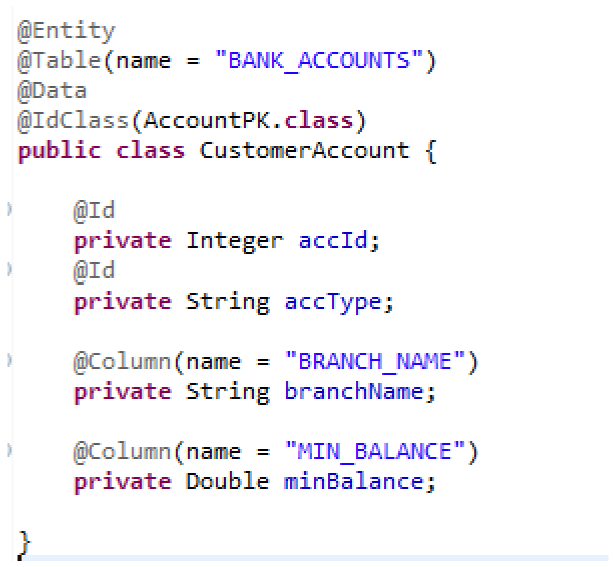
Ans: A composite primary is a combination of two or more primary keys in a database table.We can create composite primary keys in 2 ways in spring data jpa.

1. *Using @IdClass annotation – Suppose there is a CustomerAccount class which has two primary keys(account Id, account type)*
   Then we need to create an AccountPK class which must be public and should implements the serializable interface.
   Example:

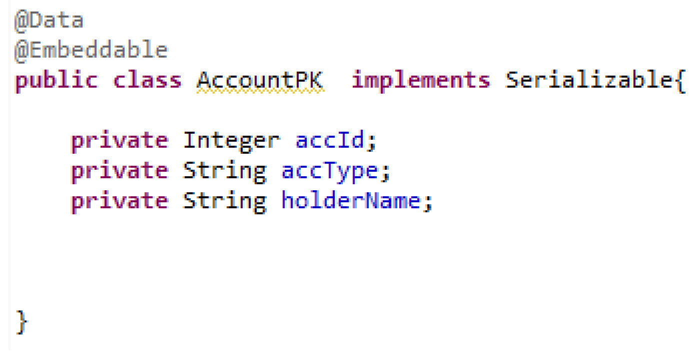
   ```
   @Data
   public class AccountPK implements Serializable {

       private Integer accId;
       private String accType;

   }
   ```

   Associate this class with the CustomerAccount Entity. In order to do that, we need to annotate the entity with the @IdClass annotation. Also the declare the primary key columns in entity with @Id annotation.



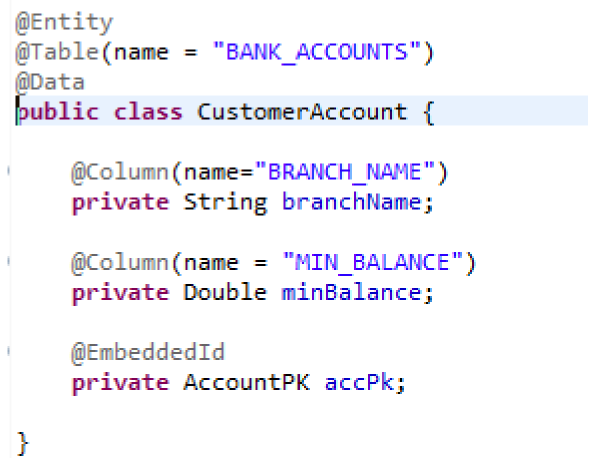
```
@Entity
@Table(name = "BANK_ACCOUNTS")
@Data
@IdClass(AccountPK.class)
public class CustomerAccount {

    @Id
    private Integer accId;
    @Id
    private String accType;

    @Column(name = "BRANCH_NAME")
    private String branchName;

    @Column(name = "MIN_BALANCE")
    private Double minBalance;

}
```

2. *Using @EmbeddedId annotation –*
   Create a class which implements Serializable interface which contains all primary keys in it. Annotate the class with @Embeddable annotation.



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

    private Integer accId;
    private String accType;
    private String holderName;



}
```

Then embed this class in entity class CustomerAccount using @EmbeddedId annotation.



```
@Entity
@Table(name = "BANK_ACCOUNTS")
@Data
public class CustomerAccount {

    @Column(name="BRANCH_NAME")
    private String branchName;

    @Column(name = "MIN_BALANCE")
    private Double minBalance;

    @EmbeddedId
    private AccountPK accPk;

}
```

Q34. What is the use of @EnableJpaRepositories method?

Ans: If the repositories classes belong to the sub package of the Spring Boot Main class, then @SpringBootApplication is enough as it scans the package using @EnableAutoConfiguration.

If the repository classes are not part of the sub package of the Main class, in that case, it will not scan the repository classes, we need to use @EnableJpaRepositories. This needs to be provided in Configuration class or SpringBootApplication class.

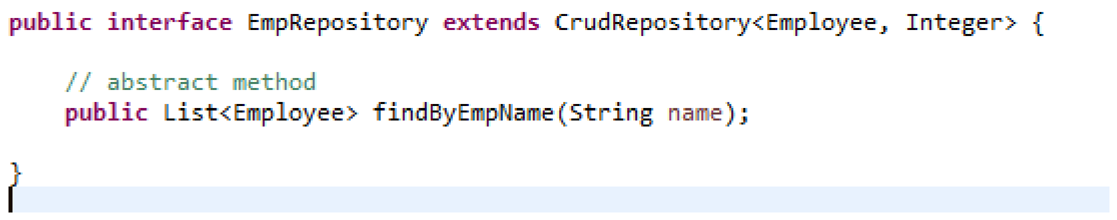
Q35. What are the rules to follow to declare custom methods in Repository.

Ans: We need to follow certain rules to declare custom methods to retrieve the data as below.

The fetch methods should start with findByXXXXX followed by property name.

Eg:

If you want to retrieve list of employees based on name, then write the custom method in Repository as:



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

    // abstract method
    public List<Employee> findByEmpName(String name);

}
```

Here the property name in entity class is empName which should be in camel case while appending to the findBy method.

Q36. Explain QueryByExample in spring data jpa.

Ans: It is another way to pass the search criteria in the where clause where the requirement is to retrieve the data based on multiple conditions.

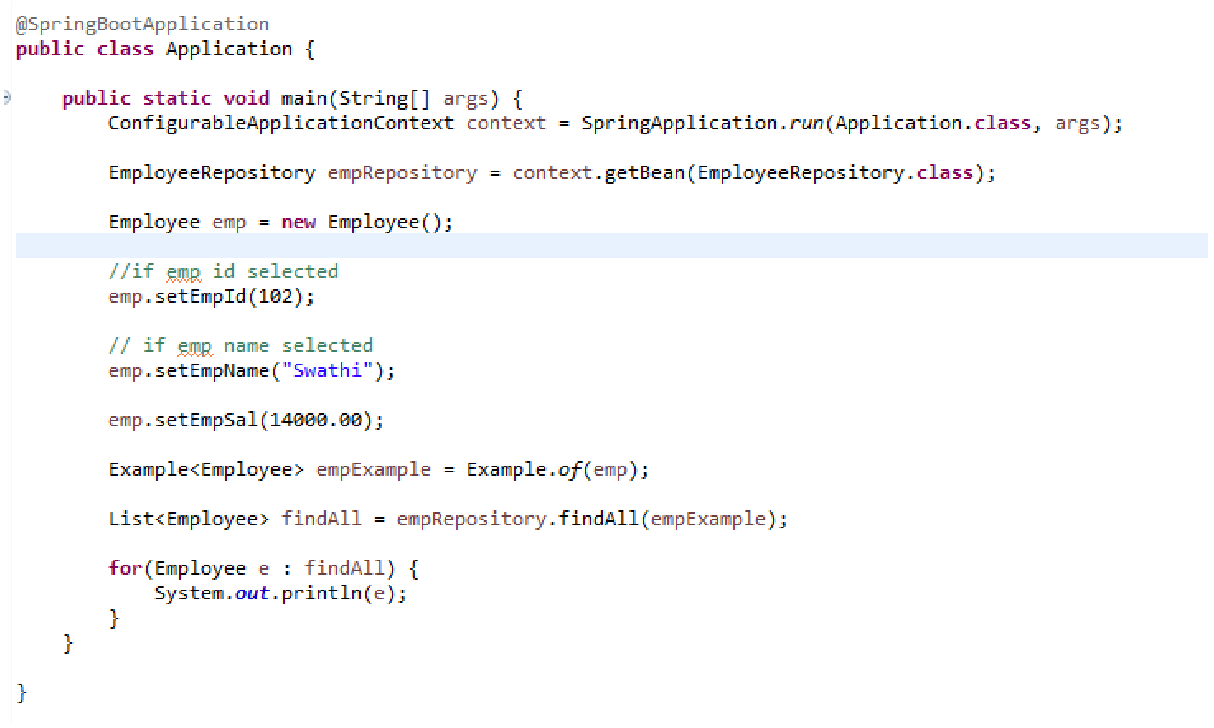
It allows us to generate the queries based on Example instance.

Example instance is created as

Example<Employee> empExample = Example.of(emp);

Where emp obj holds the search criteria.

Eg: Search for an employee whose empId is 102, name is Swathi and salary is 14000.



```java
@SpringBootApplication
public class Application {

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

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

        Employee emp = new Employee();

        //if emp id selected
        emp.setEmpId(102);

        // if emp name selected
        emp.setEmpName("Swathi");

        emp.setEmpSal(14000.00);

        Example<Employee> empExample = Example.of(emp);

        List<Employee> findAll = empRepository.findAll(empExample);

        for(Employee e : findAll) {
            System.out.println(e);
        }
    }

}
```

Q37. What is pagination and how to implement pagination in spring data?

Ans: It is the process of displaying the records in small chunks into multiple pages. Eg: in an ecommerce application, there are several products available, but all of them will not be loaded on first page, if the client clicks on second page, few of them will be loaded and so on. This is mainly to avoid the overload on the application.

Pagination contains two fields – pageSize and pageNumber.

It can be implemented using PagingAndSortingRepository which provides methods to retrieve data using pagination.

***Page findAll(Pageable pageable)** – it returns the n records based on the pageSize to be displayed on each page.*

To apply pagination on the records fetched from database, we need to create Pageable object as :

PageRequest pageReq = PageRequest.of(pgNo,pageSize);

And then pass to the find method as:

Page<Employee> pageData = repository.findAll(pageReq);

Q38. Explain few CrudRepository methods.

Ans: Some of the methods to perform DML operations are :

***findById** – to retrieve record based on the primary key.*

***findAll** – to retrieve all records from the database.*

***existsById** – to check if the record exists by passing primary key*

***count** – to check the total number of records.*

Save *– to insert a record into the database*

deleteById *– to delete a record using primaryKey*

deleteAll *– to delete all records from the table*

Q39. Difference between delete () and deleteInBatch() methods.

Ans: delete() – It is used to delete a single record at a time. It internally uses remove method of entitymanager.

deleteInBatch() – it can delete multiple records at a time, it internally calls executeUpdate() method.It is much faster than delete method.

Q40. What is the use of @Modifying annotation?

Ans: It indicates that a query method should be considered as a modifying query. It can be implied only on non-select queries (INSERT, UPDATE, DELETE). This annotation can be used only on the query methods which are defined by @Query annotation.

SPRING MVC

Q41. What is Spring MVC?

Ans: Spring MVC is one of the modules in Spring framework which helps in building web and distributed applications. It supports two design patterns

a. *MVC Design Pattern*
b. *Front Controller Design Pattern*

MVC stands for Model, View and Controller.

The major role is played by DispatcherServlet in Spring MVC which acts as a front controller which receives the incoming request and maps it to the right resource.

The main advantage of Spring MVC is that it helps in the separation of the presentation and business layer.

The components of Spring MVC are:

Model *– A model represents the data which can be an object or a group of objects.*

View *– A view represents an UI to display the data. It can be a JSP, or a Thymeleaf page.*

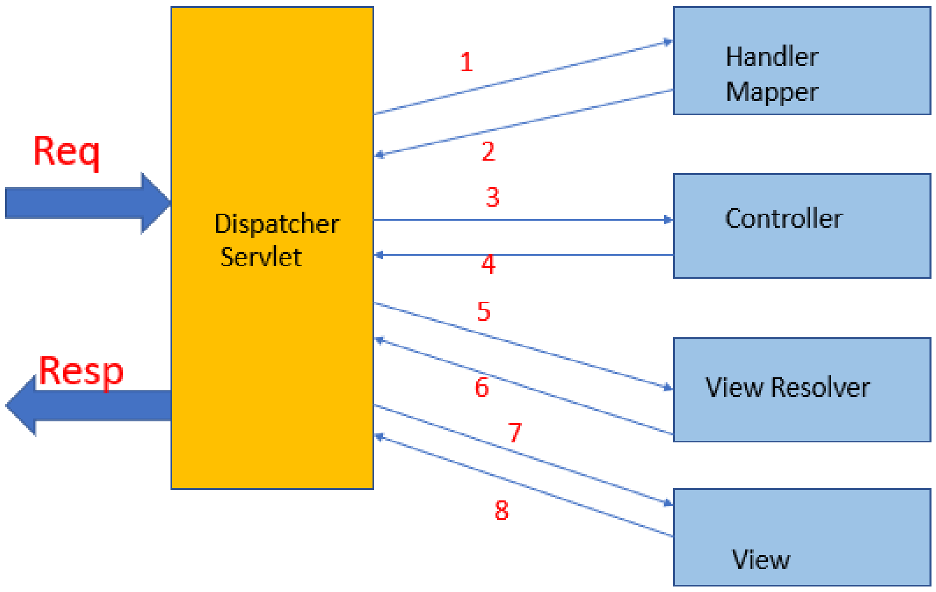
Controller *– It acts as an intermediate between model and view components and is responsible to handle the incoming requests.*

Front Controller *– Dispatcher servlet serves the main purpose of redirecting the request to the respective controller methods.*

Q42. Explain the flow of Spring MVC.

Ans: When a client request comes in, it is intercepted by the Dispatcher Servlet which acts as a Front Controller. The dispatcher servlet is responsible for pre – processing of the request and calls the handler methods to decide which controller should handle the request. It uses BeanNameUrlHandlerMapping and SimpleUrlHandlerMapping to map the request to the *corresponding controller method. The controller then processes the request and sends the response as ModelAndView back to the DispatcherServlet. Model represents the data to be displayed and view represents the component in which data should be rendered on the browser.*

Front Controller is also responsible in manipulating the response data(post-processing) before sending back to client.

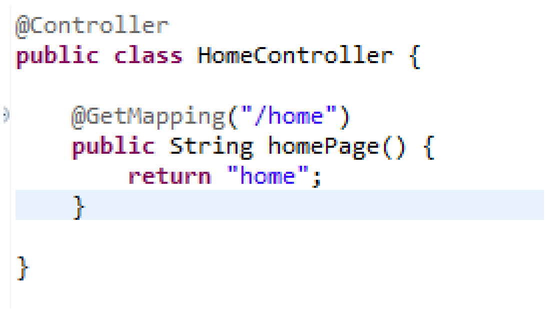


Q43. What is Dispatcher Servlet.

Ans: Dispatcher Servlet acts as a central servlet which handles all the incoming HTTP Requests and Responses. Once a client request is sent, it is received by the DispatcherServlet and it forwards the request to handler mapper to identify the corresponding controller class to handle the request. The controller performs all the business logic and hands over the response back to DispatcherServlet. The servlet then prepares the view component by looking for the view resolver in properties file and sends the data to be rendered on view page.

Q44. What is the use of @Controller annotation.

Ans: A class can be represented as a Controller class which is used to handle one or more HTTP requests. It is represented as a controller class using @Controller annotation. It is one of the stereotype annotations.

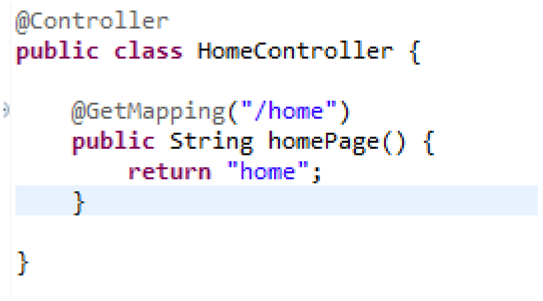


```
@Controller
public class HomeController {

    @GetMapping("/home")
    public String homePage() {
        return "home";
    }

}
```

In the example above, whenever a client sends a request with the url as ***localhost:8090/home****, the homepage method is invoked and view is returned from the method.*

Q45. What is the use of InternalResourceViewResolver?

Ans: InternalResourceViewResolver is the implementation of View Resolver interface which is used to resolve logical view names returned by the controller to a physical location where file actually exists. It is also a subclass of UrlBasedViewResolver which uses "prefix" and "suffix" to convert the logical view into physical view.



```
@Controller
public class HomeController {

    @GetMapping("/home")
    public String homePage() {
        return "home";
    }

}
```

For example, if a user tries to access /home URL and HomeController returns "home" then DispatcherServlet will check with InternalResourceViewResolver and it will use prefix and suffix to find the actual physical view.

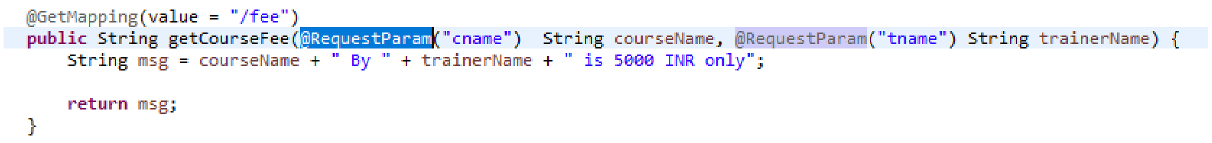
Iif prefix is "/WEB-INF/views/" and suffix is ".jsp" then "home" will be resolved to "/WEB-INF/views/home.jsp" by InternalResourceViewResolver.

Q46. Difference between @RequestParam and @PathVariable.

Ans: @RequestParam is used to access the parameter values which are passed as part of request URL.

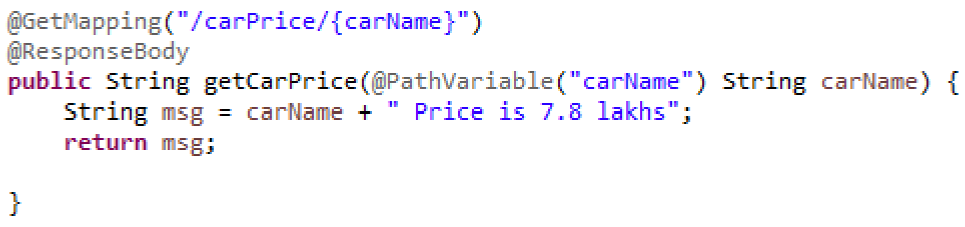
URL: ***http://localhost:8090/fee?cname=SBMS&tname=Savitha***

In the above example, we can access the parameter values of courseName and trainerName using @RequestParam annotation. In case of @RequestParam, if the parameter value is empty , it can take default value using attribute defaultValue=XXXXX



```
@GetMapping(value = "/fee")
public String getCourseFee(@RequestParam("cname")  String courseName, @RequestParam("tname") String trainerName) {
    String msg = courseName + " By " + trainerName + " is 5000 INR only";

    return msg;
}
```

@PathVariable – It is used to extract data from the request URI. Eg : if the URL is as : ***http://localhost:8090/carPrice/{carName}*** *– the value for the placeholder {carName} can be accessed using @PathVariable annotation. In order to access the carName, we need to write code as below:*



```
@GetMapping("/carPrice/{carName}")
@ResponseBody
public String getCarPrice(@PathVariable("carName") String carName) {
    String msg = carName + " Price is 7.8 lakhs";
    return msg;

}
```

Q47. Explain @Service and @Repository annotations.

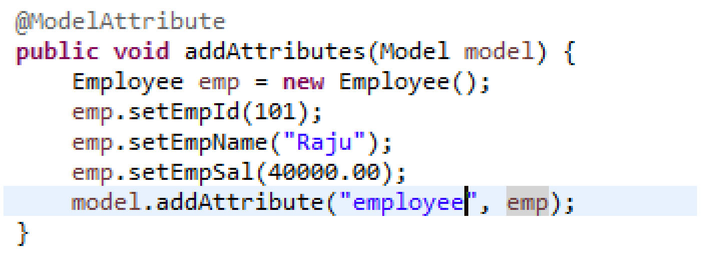
Ans: A class which is annotated with @Repository annotation is where the data is stored. It is a stereotype annotation for the persistence layer.

@Service – It indicates that a java class contains the business logic. It is also a part of @Component stereotype annotation.

Q48. What is the purpose of @Model Attribute?

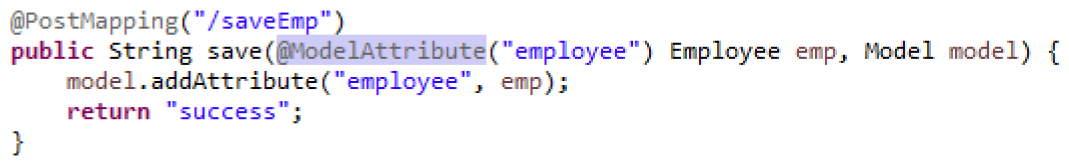
Ans: It is part of Spring MVC module and can be used in two scenarios:

@ModelAttribute at method level: *When used at method level, it indicates that a method will return one or more model attributes.*



```
@ModelAttribute
public void addAttributes(Model model) {
    Employee emp = new Employee();
    emp.setEmpId(101);
    emp.setEmpName("Raju");
    emp.setEmpSal(40000.00);
    model.addAttribute("employee", emp);
}
```

@ModelAttribute at method argument: *When it is used at method argument, it indicates that the argument should be retrieved from the form data. It binds the form data to the bean object.*

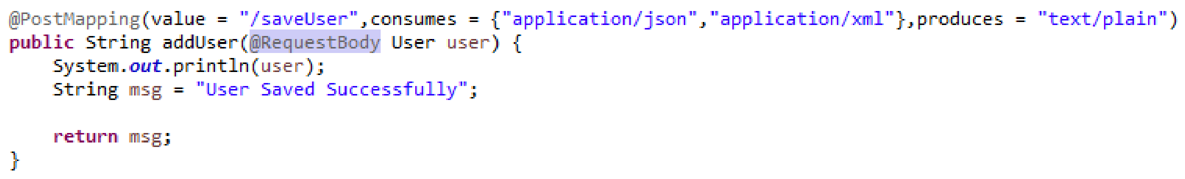


```
@PostMapping("/saveEmp")
public String save(@ModelAttribute("employee") Employee emp, Model model) {
    model.addAttribute("employee", emp);
    return "success";
}
```

Q49. Explain @RequestBody annotation.

Ans: This annotation indicates that the method parameter should be bound to the body of the HTTP request.

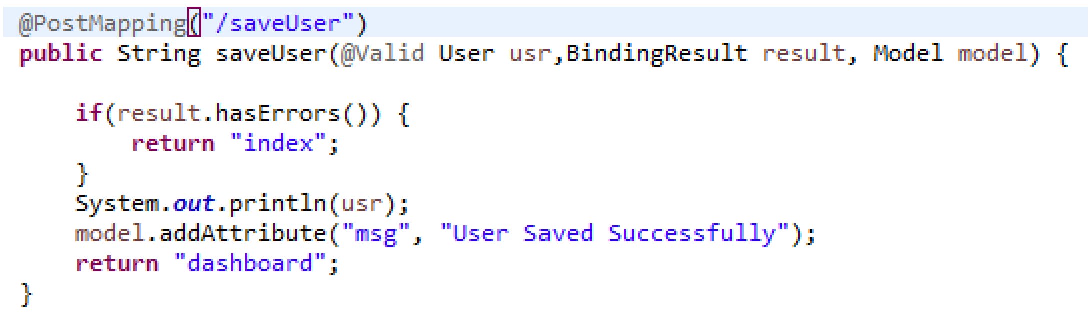
Eg:



```
@PostMapping(value = "/saveUser",consumes = {"application/json","application/xml"},produces = "text/plain")
public String addUser(@RequestBody User user) {
    System.out.println(user);
    String msg = "User Saved Successfully";

    return msg;
}
```

Q50. What is the use of Binding Result.

Ans: BindingResult holds the result of the validation and binding and contains errors that have occurred. The BindingResult is a spring's object which must come right after the model object that is validated or else Spring will fail to validate the object and throw an exception.



```
@PostMapping("/saveUser")
public String saveUser(@Valid User usr,BindingResult result, Model model) {

    if(result.hasErrors()) {
        return "index";
    }
    System.out.println(usr);
    model.addAttribute("msg", "User Saved Successfully");
    return "dashboard";
}
```

Q51. How does Spring MVC support for validation.

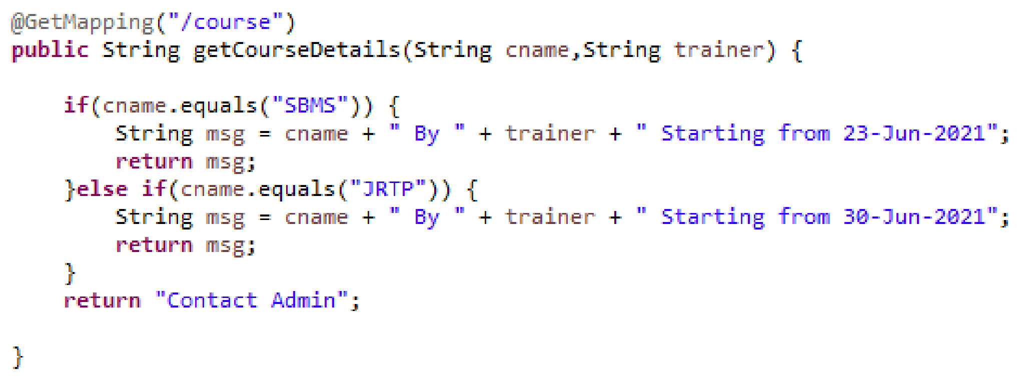
Ans: It is used to restrict the user input provided by the user. Spring provides the validation API where the BindingResult class is used to capture the errors raised while validating the form.

We need to add spring-boot-starter-validation in pom.xml.

Q52. Explain @GetMapping and @PostMapping.

Ans: @GetMapping is an alternative for @RequestMapping (method = RequestMethod.GET)

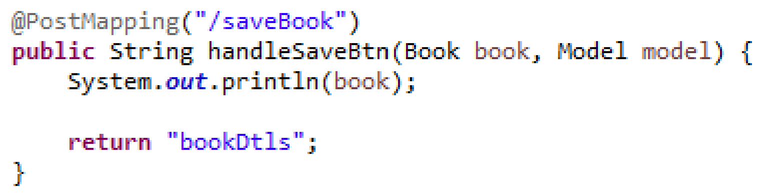
It handles the HTTP Get methods matching with the given URI.



```
@GetMapping("/course")
public String getCourseDetails(String cname,String trainer) {

    if(cname.equals("SBMS")) {
        String msg = cname + " By " + trainer + " Starting from 23-Jun-2021";
        return msg;
    }else if(cname.equals("JRTP")) {
        String msg = cname + " By " + trainer + " Starting from 30-Jun-2021";
        return msg;
    }
    return "Contact Admin";

}
```

@ PostMapping is an alternative for @RequestMapping (method = RequestMethod.POST)

It handles the HTTP Post methods matching with the given URI.



```
@PostMapping("/saveBook")
public String handleSaveBtn(Book book, Model model) {
    System.out.println(book);

    return "bookDtls";
}
```

Q53. How to send the data from Controller to UI.

Ans: In spring mvc, controller is responsible to send the data to UI. We have Model object and ModelAndView to send the data from controller to UI.

The data in the model object is represented in key-value format as below.

model.addAttribute("key","value");

```
@GetMapping("/greet")
public String getWishMsg(Model model) {

     model.addAttribute("msg", "God Bless You");
     return "index";
}
```

Q54. What is the purpose of query parameter?

Ans: QueryParameters are used to send the data from UI to Controller.The query parameters are passed in the URL and starts with ?.

Multiple query parameters will be represented with ampersand operator (&).

Query Parameters are appended at the end of the URL and this can be retrieved from the url using @RequestParam annotation.

Eg: http://localhost:8090/studentapp?sid=100&sname=Raju

Q55. Describe the annotations to validate the form data.

Ans: Some annotations to validate the form data.

@NotNull – It is used to check if the value entered is not null.

@NotEmpty – It checks whether the annotated element is not null nor empty.

@Email – It checks whether the given value is a valid email address.

@Size – It is used to determine that size of the value must be equal to the mentioned size.

@Null – It checks that the value is null.

Q56. What do you know about Thymeleaf?

Ans: Thymeleaf is used to design the presentation logic. It is a java template engine that helps in processing and creating HTML,javascript,etc.

It reads the template file and parses it and produces web content directly on the browser.

It helps in developing dynamic web content.

In spring mvc, we need to add the dependency as

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

Q57. Explain the use of @ResponseBody annotation.

Ans: @ResponseBody on a method indicates that the return type should be directly written to the response object.

```java
@GetMapping("/plandata")
@ResponseBody
public InsurancePlan getPlanData() {
    InsurancePlan plan = new InsurancePlan();
    plan.setPlanId(101);
    plan.setPlanName("Jeevan Anand");
    plan.setPlanStatus("Approved");

    return plan; //method return type is object
}
```

Q58. What is the role of Handler Mapper in Spring MVC.

Ans: Handler mapper is used to map the incoming request to the respective controller method.DispatcherServlet forwards the request received to handler mapper.By default ,It uses BeanNameUrlHandlerMapping and DefaultAnnotationHandlerMapping to map the request to the controller.

Q59. How will you map the incoming request to a Controller method.

Ans: When a client request is received by the dispatcher servlet with the URI , for example as ***http://localhost:8090/home*** *, the central servlet forwards the request to handler mapper which checks for the controller method which matches the url pattern, in this case /home and returns the name of the controller. The front controller then sends the request to the appropriate Controller which processes the business logic and sends back the response to the client.*

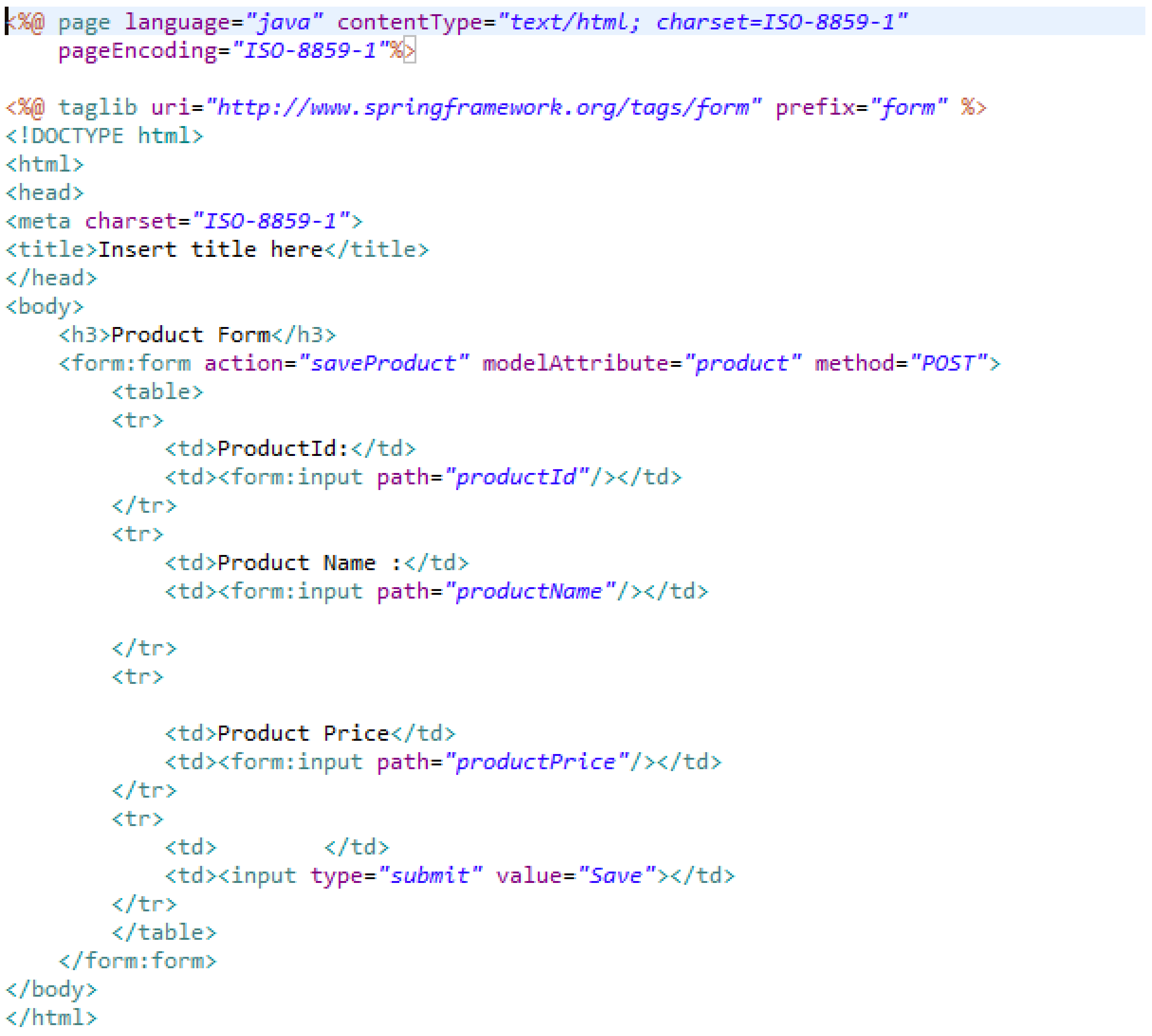
Q60. How to bind the form data to Model Object in Spring MVC.

Ans: In order to create a form in spring, we need to use <form:form> tag. Eg: to store the form data into model object.

Create a POJO class:

```java
@Data
public class Product {

    private Integer productId;

    private String productName;

    private Double productPrice;


}
```

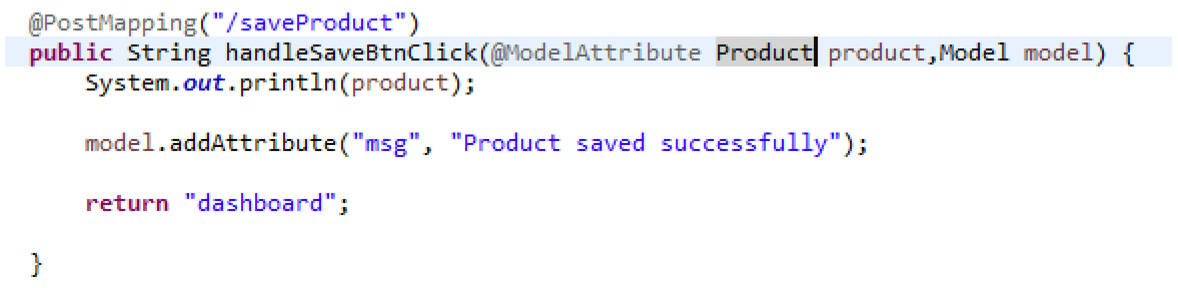
Create a product.jsp which contains the form fields using spring mvc form tag library.



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h3>Product Form</h3>
    <form:form action="saveProduct" modelAttribute="product" method="POST">
        <table>
        <tr>
            <td>ProductId:</td>
            <td><form:input path="productId"/></td>
        </tr>
        <tr>
            <td>Product Name :</td>
            <td><form:input path="productName"/></td>

        </tr>
        <tr>

            <td>Product Price</td>
            <td><form:input path="productPrice"/></td>
        </tr>
        <tr>
            <td>        </td>
            <td><input type="submit" value="Save"></td>
        </tr>
        </table>
    </form:form>
</body>
</html>
```

In order to bind the form data , we have an attribute called as "modelAttribute", which specifies the name of the bean class to which form data should be binded.

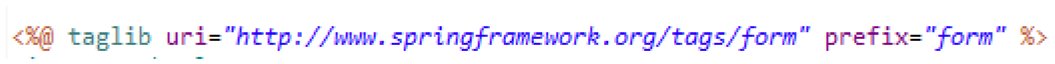
The form data can be retrieved in the controller class using @ModelAttribute annotation as below. The modelAttribute object specified in the JSP should match with the one in controller class, else form binding won't work.



```
 @PostMapping("/saveProduct")
 public String handleSaveBtnClick(@ModelAttribute Product product,Model model) {
     System.out.println(product);

     model.addAttribute("msg", "Product saved successfully");

     return "dashboard";

 }
```

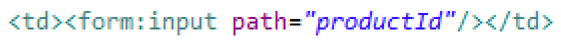
Q61. What is Spring MVC Tag library.

Ans: Spring provides a tag library which is used in creating view component. It provides tags to create HTML fields, error messages, etc. It is a predefined library which can be used in JSP by using the tag as:



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

After adding the above tag, we can create HTML form input text by using prefix as "form".



```
<td><form:input path="productId"/></td>
```

Q62. Difference between @Controller and @RestController annotations.

Ans: A class which is annotated with @Controller indicates that it is a controller class which is responsible to handle the requests and forwards the request to perform business logic. It returns a view which is then resolved by ViewResolver after performing business operations.

@RestController is used in REST Webservices and is combination of @Controller and @ResponseBody.It returns the object data directly to HTTP Response as a JSON or XML.

Q63. Explain few tags in Spring MVC tag library.

Ans: Few of the spring mvc tags are :

<form:form> - It is used to create a HTML form.

<form:input> - It is used to create input text field.

<form:radiobutton> - It is used to create a radio button.

<form:select> - It is used to create a dropdown list.

<form:hidden> - It is used to create a hidden field.

<form:checkbox> - It is used to create a checkbox.

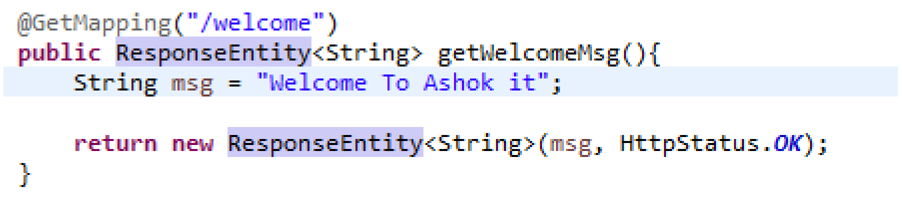
<form:option> - It is used to create a single Html option inside select tag.

Q64. Explain the use of @ResponseEntity annotation.

Ans: @ResponseEntity is used to represent the entire HTTP response such as status code, headers and response body.

We can return the entire response from the endpoint. When using @ResponseBody , it returns the value into the body of the http response.

Example:



```
@GetMapping("/welcome")
public ResponseEntity<String> getWelcomeMsg(){
    String msg = "Welcome To Ashok it";

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

Q65. How to handle exceptions in Spring MVC.

Ans: In Spring Boot, exceptions can be handled using below two annotations:

- *@ExceptionHandler -specific to a controller class*
- *@ControllerAdvice – common to all controllers.*

Q66. Explain @ControllerAdvice in Spring Boot.

Ans: @ControllerAdvice is a specialization of @Component annotation which is used to handle the exceptions across the whole application by providing a global code which can be applied to multiple controllers.

Spring Actuator

Q67. What is Spring actuator and its advantages.

Ans: An actuator is mainly used to provide the production ready features of an application. It helps to monitor and manage our application. It provides various features such as healthcheck, auditing, beans loaded into the application,etc.

Q68. How will you enable actuator in spring boot application.

Ans: An actuator can be enabled by adding the starter pom into the pom.xml.

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

Q69. What are the actuator endpoints which are needed to monitor the application.

Ans: Actuators provide below pre-defined endpoints to monitor our application.

- *Health*
- *Info*
- *Beans*
- *Mappings*
- *Configprops*
- *Httptrace*
- *Heapdump*
- *Threaddump*
- *Shutdown*

Q70. How to get the list of beans available in spring application.

Ans: Spring Boot Actuator provides an endpoint url /beans to load all the spring beans of the application.

Q71. How to enable all endpoints in actuator?

Ans: In order to expose all endpoints of actuator, we need to configure it in application properties/yml file as:

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

Q72. What is a shutdown in the actuator?

Ans: A shutdown is an endpoint that helps application to shut down properly. This feature is not enabled by default. We can enable it by giving the below command in properties file.

management.endpoint.shutdown.enabled=true

Spring Security

Q73. What is Spring Security?

Ans: Spring security is a powerful access control framework. It aims at providing authentication and authorization to java applications. It enables the developer to impose security restrictions to save from common attacks.

Q74. What are the features of Spring Security?

Ans: Spring security provides many features as below:

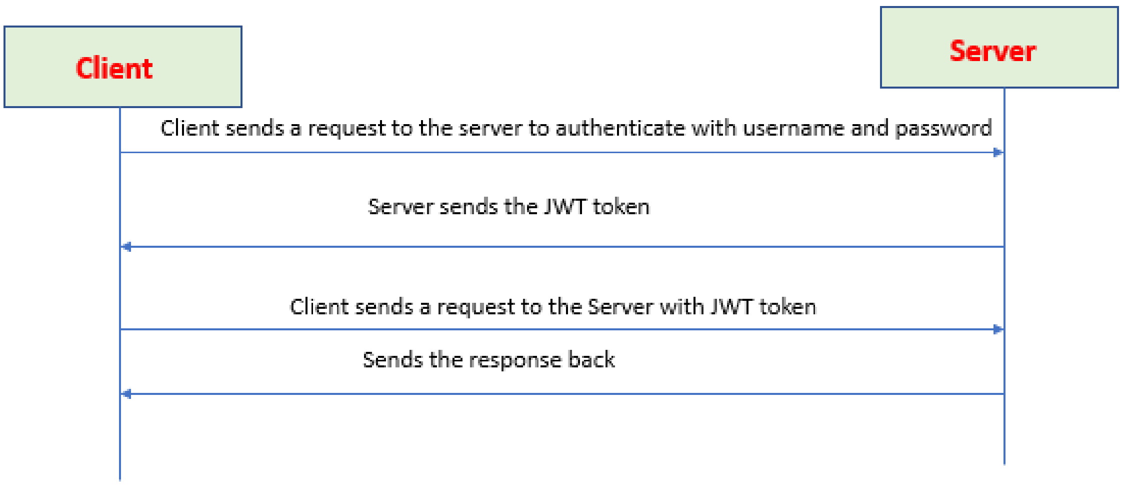
- *Authentication and Authorization.*
- *Supports Basic and Digest Authentication.*
- *Supports CSRF Implementation.*
- *Supports Single Sign-on.*

Q75. How to implement JWT?

Ans: JWT stands for Json Web Token which helps in implementing token-based security. Token is generated using the secret key. We need to add below dependency in pom.xml.

```
<dependencies>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt</artifactId>
            <version>0.9.1</version>
        </dependency>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.0</version>
        </dependency>
    </dependencies>
```

The following diagram depicts the working of JWT.


Client
Server
Client sends a request to the server to authenticate with username and password
Server sends the JWT token
Client sends a request to the Server with JWT token
Sends the response back

Q76. What is DelegatingFilterProxy in Spring Security.

Ans: It is a predefined filter class which helps in performing pre-processing of the request.It supports for both authentication and authorization. It is a proxy servlet filter which acts as an intermediator before redirecting the request to dispatcher servlet.

Q77. What is Spring Security OAuth2.

Ans: OAuth2 is an authorization framework, granting clients access to protected resources via authorization server. It allows end user's account information to be used by third party services(eg.facebook) without exposing user's password.

The oAuth token are random strings generated by the authorization server.

There are 2 types of token.

Access token – It is sent with each request, usually valid for about an hour only.

Refresh token – It is used to get a new access token, not sent with each request. It lives longer than access token.

Q78. What is the advantage of using JWT Token?

Ans: Advantages of using JWT Token:

- *The jwt token has authentication details and expire time information.*
- *It is one of the approaches to secure the application data because the parties which are interacting are digitally signed.*
- *It is very small token and is better than SAML token.*
- *It is used at internet scale level, so it is very easy to process on user's device.*

Q79.What is authentication?

Ans: Authentication is the mechanism to identify whether user can access the application or not.

Q80.What is authorization?

Ans: Authorization is the process to know what the user can access inside the application and what it cannot i.e which functionality it can access and which it cannot.

Q.81. What is filter Chain Proxy?

Ans: The filter Chain Proxy contains multiple security filter chains and a task is delegated to the filter chain based on the URI mapping. It is not executed directly but started by DelegatingFilterProxy.

Q.82. What is security context in Spring Security.

Ans: It is used to store the details of the current authenticated user, which is known as principle. So if we want to get the current username, we need to get the SecurityContext.

Q.83. Difference between has Authority and hasRole?

Ans: hasRole – defines the role of the user.It does not use the ROLE_prefix but it will automatically added by spring security as hasRole(ADMIN);

has Authority – defines the rights of the user. It uses ROLE prefix while using has Authority method as has Authority (ROLE_ADMIN)

Q.84. How to enable spring boot security in spring boot project?

Ans: If Spring boot Security dependency is added on class path, it automatically adds basic authentication to all endpoints. The Endpoint "/" and "/home" does not require any authentication. All other Endpoints require authentication.

The following dependency needs to be added.

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-security</artifactId>

</dependency>
```

Q.85.What is Basic Authentication?

Ans: In Basic Authentication, we send username and password as part of the request to allow user to access the resource. The user credentials are sent as authorization request headers. This approach is used to authenticate the client requests.

Q.86.What is Digest Authentication?

Ans: Digest Authentication is more preferable when compared to basic authentication as the credentials are in encrypted format by applying hash function to username,password,etc. It does not send the actual password to the server. Spring security provides digest authentication filter to authenticate the user using digest authentication header.

Q.87. How to get current logged in user in spring security.

Ans: We can get the current logged in user by using the following code snippet.

```
User user = (User)SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

String name = user.getUsername();

Q.88.What is SSL and its use?

Ans: SSL stands for secure socket layer which is an encryption- based internet security protocol.

It is mainly used to secure client information (such as credit card number /password/ssn) to a web server.

SSL provides a secure channel between two machines or devices running over the internet. A common example is when SSL is used to secure communication between a web browser and a web server. This changes the address of the website from HTTP to HTTPS, basically 'S' stands for 'Secure'.

Q.89. What is salting?

Ans: Salting is used to generate random bytes that is hashed along with the password. Using salt , we can add extra string to password , so that hackers finds difficult to break the password. The salt is stored as it is, and need not be protected.

Q90.What is hashing in spring security.

Ans: Hashing is an approach where a string is converted into encoded format using hashing algorithm. The hashing algorithm takes input as password and returns hashed string as output. This hashed data is stored in the database instead of plain text which is easily vulnerable to hacker attacks.

Q91.How to secure passwords in a web application?

Ans: Generally, passwords should not be stored as plain text into the storage as it can easily be accessed by the hackers.

We need to use encryption techniques before storing the password.

We need to use hashing and salting techniques to prevent from security breaches.

Q92.What is AuthenticationManager in spring security?

Ans: Authentication manager is the interface that provides the authentication mechanism for any object. The most common implementation of it is the AuthenticationProvider.

If the principal of the input is valid , and authenticated , it returns an authentication instance if successful.

It checks whether the username and password is authenticated to access a specific resource.

Q93. What are the various ways to implement security in spring boot project?

Ans: There are 3 ways to implement security.

a. *In Memory Credential Security*
b. *Using Database*
c. *Using UserDetailsService*

In Memory Credential – *In this mechanism, we configure the user credentials in the application itself, and use it when there is a request to validate the user.*

Using JDBC Credentials *– Here, the user credentials are stored into the database and when the client request comes, it is validated against it.*

Using UserDetailsService *– It is an interface provided by Spring framework. After entering the username in the form and clicking on Login button invokes a call to this service. It locates the user based on the username provided. It contains a method loadUserByUsername(String username) which returns UserDetails object.*

General :

Q94. What are the essential components of Spring Boot?

Ans: Some of the components of Spring Boot are:

- *Spring Boot Starter*
- *Spring Boot autoconfiguration*
- *Spring Boot Actuator*
- *Spring Boot CLI*

Q95. What is the use of profiles in Spring Boot?

Ans: Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. For eg, if we want to enable swagger configuration only in QA environment, it can be done using spring profiles.

Q96. How can you set active profile in Spring Boot.

Ans: We can set the active profile by using configuration properties as:

spring.profiles.active=production

Q95. What is AOP?

Ans: Aspect Oriented Programming aims at separating the cross-cutting logics from the main business logic.

Q96. What is YAML?

Ans: YAML is mainly used for configuration purpose. It is similar to properties file and provides more readability.

Q99. Use of @Profile annotation.

Ans: The @Profile annotation indicates that a component is eligible for registration when the specified profile is active. The default profile is called default, all the beans that do not have a profile set belong to this profile.

Q100. How to get current profile in Spring Boot.

Ans: String[] activeProfiles = environment. getActiveProfiles();