

Unit-5

Spring Framework

Spring Framework

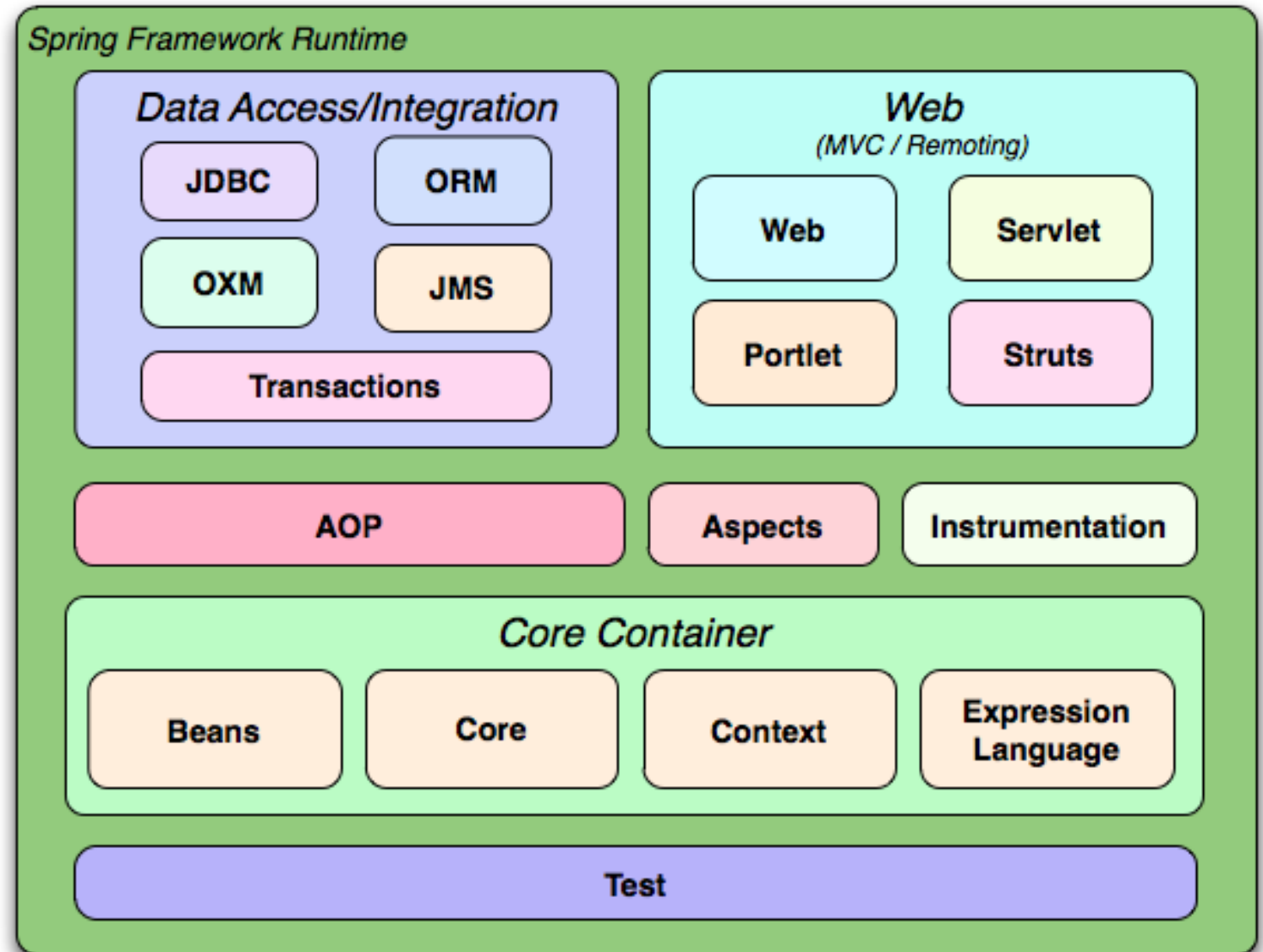
- Open-Source Framework
- Standalone and Enterprise application can be developed
- Released in 2003(initial), 2004(production) developed by Rod Johnson
- Spring is a *lightweight* framework.
- It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc.
- The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.
- The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

Advantages

- Modular and lightweight(lightweight and easy to maintain applications)
- Flexible configuration(supports Java-based, XML-based and annotation-based configurations)
- Dependency Injection(dependency management)
- Aspect oriented programming(Allows developers to separate code from the features like logging, transactions, security etc.)
- Easy database handling(reduce boilerplate code increase efficiency)
- Testing support
- Security(robust framework for implementing authentication, authorization)
- High integration capabilities(with other frameworks and technologies like angular, react, JMS, SOAP, REST)
- High Scalability
- Open-Source

Modules

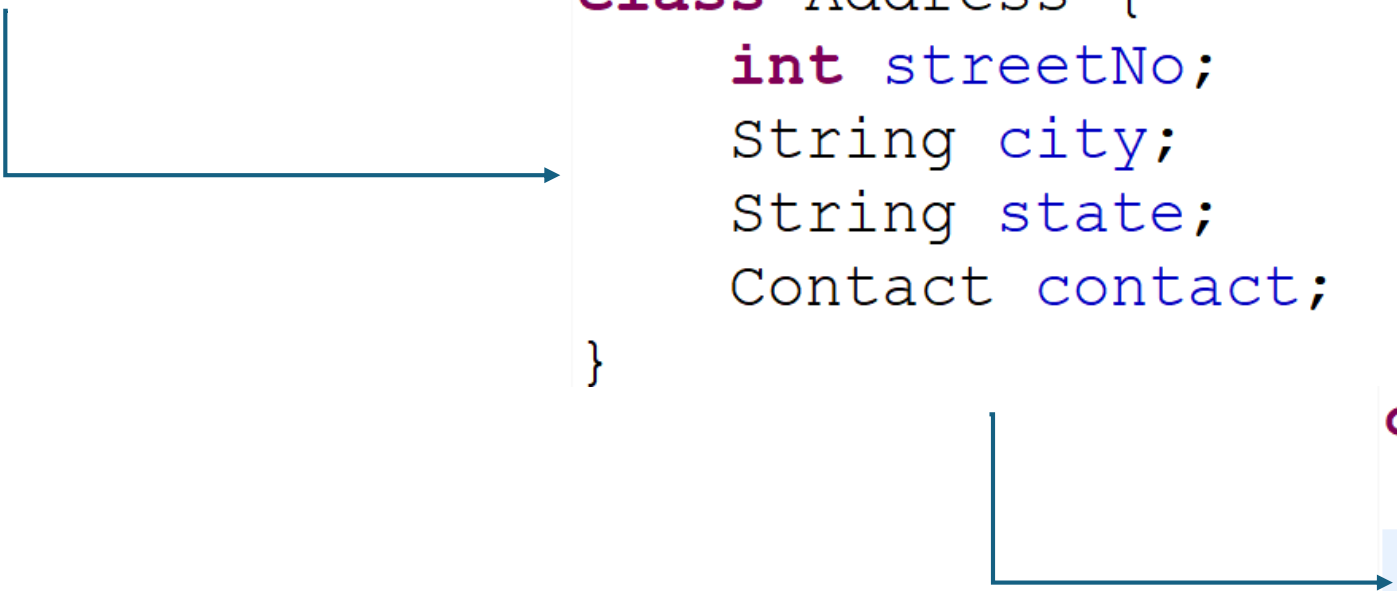
The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



Dependency?

```
class Person {  
    int adhaar;  
    String Name;  
    Address address;  
}
```

- **Example of dependency**
- Code has very high degree of coupling due to aggregation
- To create Object of class Person, we depend on Address Object, and to create Address object, we need contact



```
class Address {  
    int streetNo;  
    String city;  
    String state;  
    Contact contact;  
}
```

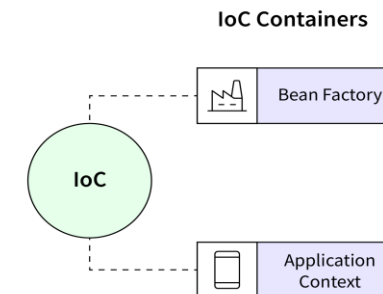
```
class Contact {  
    double mob;  
    String email;  
}
```

Spring Container

- The Spring container is the **core of the Spring Framework**.
- Manages Bean Objects(create, initialize, destroy)[Life cycle of bean]
- It is responsible for creating, configuring, and managing the objects that make up your application.
- The container uses a technique called **dependency injection** to manage the relationships between objects.
- Transaction Management

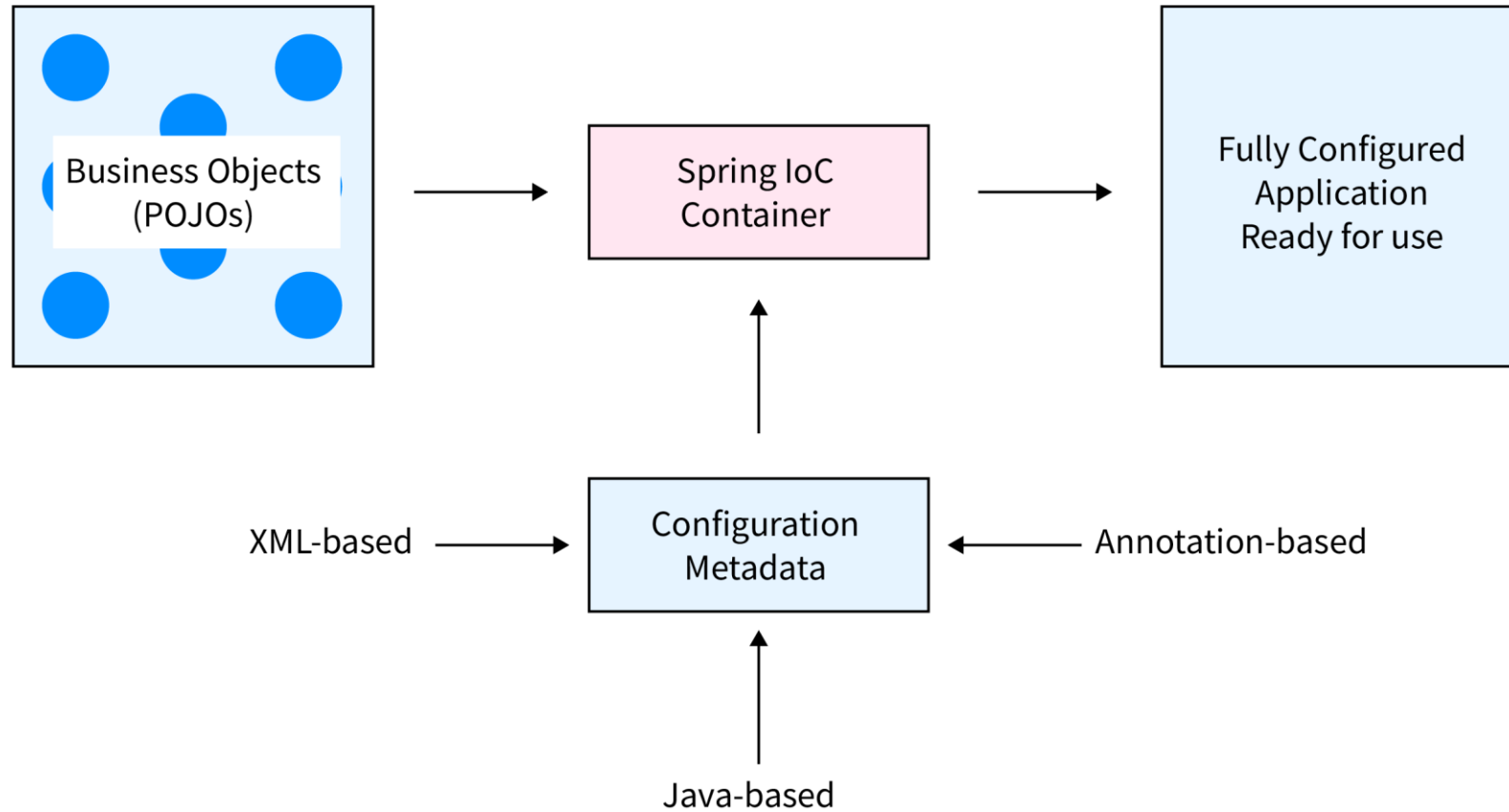
Spring container are of **TWO TYPES**

1. BeanFactory(old Method)
2. ApplicationContext(**new Method**)

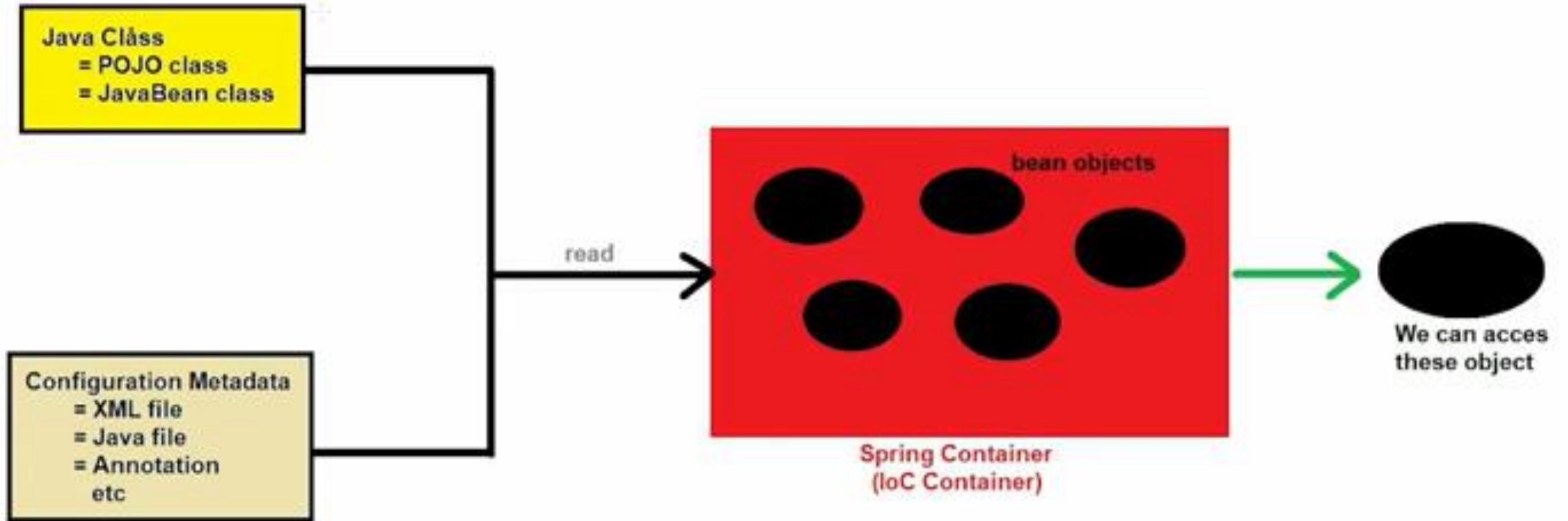


Working of Spring Container

Spring IoC Container



Working of Spring Container



Working of Spring Container

- = The ApplicationContext (Spring Container) is an interface in Spring which is used to manage beans, handle application events, and access resources
- = Some implemented classes of ApplicationContext are :-
 1. ClassPathXmlApplicationContext (used for XML configurations)
 2. AnnotationConfigApplicationContext (used for Java configurations)etc

Spring Framework Example with Java Configuration file

```
package in.cs.beans;
public class Employee {
    String name;
    int eid;
    String email;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getEid() {
        return eid;
    }
    public void setEid(int eid) {
        this.eid = eid;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public void display() {
        System.out.println("Name: "+name);
        System.out.println("Id: "+eid);
        System.out.println("Email: "+email);
    }
}
```

```
package in.cs.resources;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import in.cs.beans.Employee;

@Configuration
public class JavaCofigurationFile {

    @Bean
    public Employee emp() {
        Employee e=new Employee();
        e.setEid(1);
        e.setName("Dr.Mohit");
        e.setEmail("mohit.agarwal@gmail.com");
        return e;
    }
}
```

```
package in.cs.main;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import in.cs.resources.JavaCofigurationFile;
import in.cs.beans.Employee;

public class Main{
    public static void main(String []args) {
        ApplicationContext context=new
        AnnotationConfigApplicationContext(JavaCofigurationFile.class);
        Employee e=(Employee)context.getBean("emp");
        e.display();
    }
}

//add a jar file aop
```

Inversion of Control (IoC)

- Inversion of Control (IoC) is a **design principle** that emphasizes keeping Java classes independent of each other.
- IoC is achieved through **Dependency Injection (DI)**.
- IoC refers to transferring the control of objects and their dependencies from the main program to a container or framework.
- The IoC container uses two primary mechanisms to work:

Bean instantiation:

- The IoC container is responsible for creating and configuring beans. This can be done through **XML configuration, Java annotations, or a combination of both.**

Dependency injection:

- The IoC container injects dependencies into beans. This means that the IoC container is responsible for providing beans with the objects they need to function.

Spring Dependency Injection

- Dependency Injection (DI) is a **design pattern** that allows you to decouple your code by making it easier to create and manage objects.
- In Spring, DI is implemented using the Inversion of Control (IoC) container. The IoC container is responsible for creating and managing objects, and it injects them into your code when needed.
- **Dependency Injection** is a fundamental aspect of the Spring framework, through which the Spring container “*injects*” objects into other objects or “*dependencies*”.

There are two types of Spring Dependency Injection.

- **Setter Dependency Injection (SDI)**
- **Constructor Dependency Injection (CDI)**

Spring IoC (Inversion of Control)	Spring Dependency Injection
Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle.	Spring Dependency injection is a way to inject the dependency of a framework component by the following ways of spring: Constructor Injection and Setter Injection
Spring helps in creating objects, managing objects, configurations, etc. because of IoC (Inversion of Control).	Spring framework helps in the creation of loosely-coupled applications because of Dependency Injection.
Spring IoC is achieved through Dependency Injection.	Dependency Injection is the method of providing the dependencies and Inversion of Control is the end result of Dependency Injection.
IoC is a design principle where the control flow of the program is inverted.	Dependency Injection is one of the subtypes of the IOC principle.
<u>Aspect-Oriented Programming</u> , <u>Dependency look up are other ways</u> to implement Inversion of Control.	In case of any changes in business requirements, no code change is required.

Dependency Injection using Setter Method

```
package in.cs.beans;
public class Student {
    private String name;
    private int roll;
    private Address address;
    public void setName(String name) {
        this.name = name;
    }
    public void setRoll(int roll) {
        this.roll = roll;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
    public void display() {
        System.out.println("name"+name);
        System.out.println("roll"+roll);
        System.out.println("Address"+address);
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <!-- bean definitions here -->
    <bean class="in.cs.beans.Address" id="AddId">
        <property name="houseNo" value="103"></property>
        <property name="city" value="Ghaziabad"></property>
        <property name="state" value="UP"></property>
        <property name="pin" value="44556677"></property>
    </bean>
    <bean class="in.cs.beans.Student" id="stdId">
        <property name="name" value="Mohit"></property>
        <property name="roll" value="101"></property>
        <property name="address" ref="AddId"></property>
    </bean>
</beans>
```

```
package in.cs.beans;

public class Address {
    private int houseNo;
    private String city;
    private String state;
    private int pin;
    public void setHouseNo(int houseNo) {
        this.houseNo = houseNo;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public void setState(String state) {
        this.state = state;
    }
    public void setPin(int pin) {
        this.pin = pin;
    }
    @Override
    public String toString() {
        String add="@"+houseNo+"-"+city+"-"+state+"-"+pin;
        return add;
    }
}
```

```
package in.cs.container;
import in.cs.beans.Student;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainDI {
    public static void main(String []args) {
        String
        configFile="/in/cs/resource/applicationContext.xml";
        ApplicationContext context=new
        ClassPathXmlApplicationContext(configFile);
        Student s=(Student)context.getBean("stdId");
        s.display();
    }
}
```


// Dependency Injection Using Constructor

```
package in.cs.beans;

public class Student {
    private String name;
    private int roll;
    private Address address;
    public Student(String name, int roll, Address address)
    {
        this.name=name;
        this.roll=roll;
        this.address=address;
    }
    public void display() {
        System.out.println("name"+name);
        System.out.println("roll"+roll);
        System.out.println("Address"+address);
    }
}
```

```
package in.cs.container;
import in.cs.beans.Student;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationCon
text;
public class MainDI {
    public static void main(String []args) {
        String
        configFile="/in/cs/resource/applicationContext.xml";
        ApplicationContext context=new
        ClassPathXmlApplicationContext(configFile);
        Student s=(Student)context.getBean("stdId");
        s.display();
    }
}
```

```
,
package in.cs.beans;

public class Address {
    private int houseno;
    private String city;
    private String state;
    private int pin;
    public Address(int houseno, String city, String state, int pin)
    {
        this.houseno=houseno;
        this.city=city;
        this.state=state;
        this.pin=pin;
    }
    @Override
    public String toString() {
        String add="@"+houseno+"-"+city+"-"+state+"-"+pin;
        return add;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <!-- bean definitions here -->
    <bean class="in.cs.beans.Address" id="AddId">
        <constructor-arg value="111"></constructor-arg>
        <constructor-arg value="Vaishali"></constructor-arg>
        <constructor-arg value="UP"></constructor-arg>
        <constructor-arg value="111777"></constructor-arg>
    </bean>
    <bean class="in.cs.beans.Student" id="stdId">
        <constructor-arg value="Rohit"></constructor-arg>
        <constructor-arg value="1"></constructor-arg>
        <constructor-arg ref="AddId"></constructor-arg>
    </bean>
</beans>
```

Setter Method DI	Constructor DI
Dependencies are injected into class through setter methods	Dependencies are injected into a class through constructor
More readable	Less readable
More Flexible	Less Flexible

Maven

Project build tool

Automates build
process



->Clean

->Compile

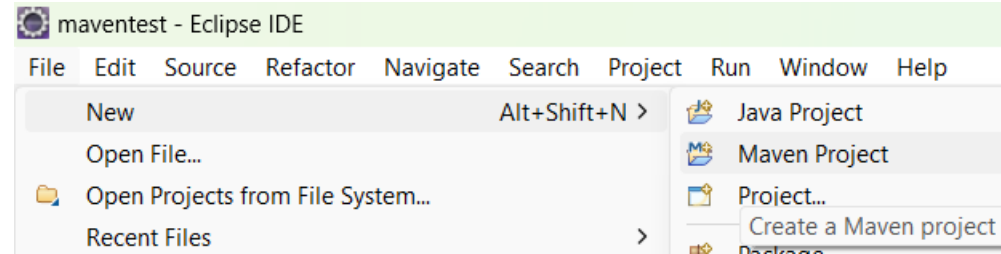
->test

->package

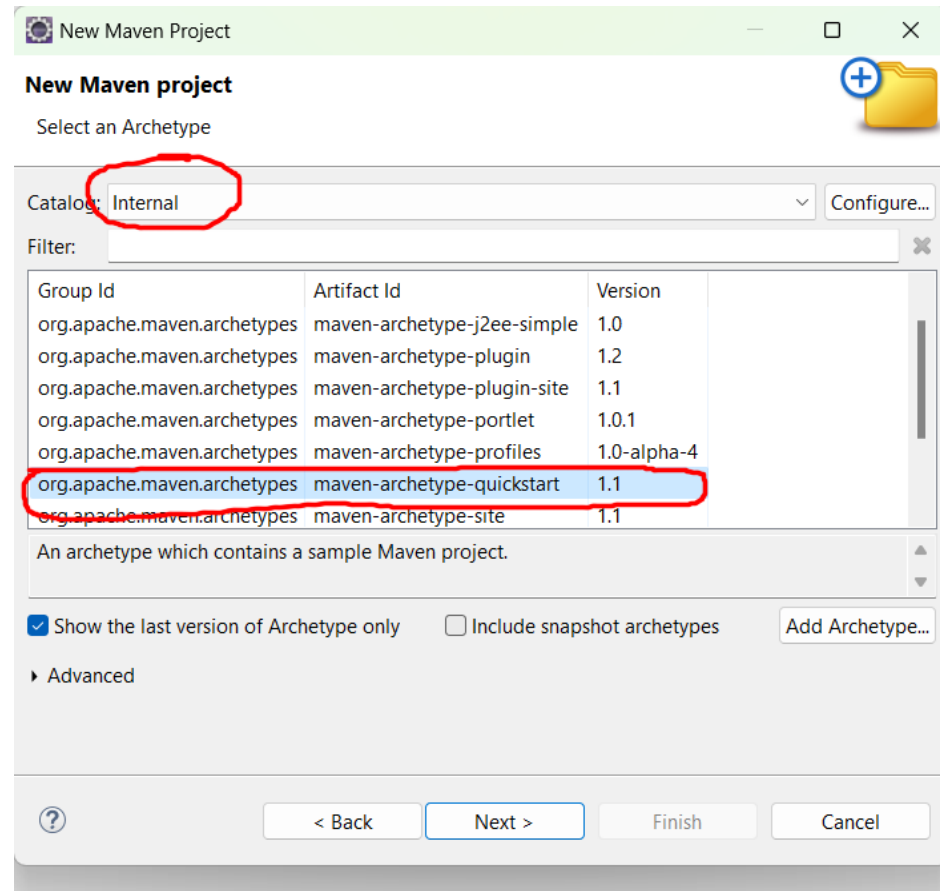
->install

->deploy

1. Create Maven project



2. Select Catalog and archetype as marked



3. Provide group id and artifact Id

4. Press finish

*(Internet must be on)

5. Open pom.xml(project object model)

New Maven Project

New Maven project

Specify Archetype parameters

Group Id: com.test

Artifact Id: DIDemo

Version: 0.0.1-SNAPSHOT

Package: com.test.DIDemo

☒ run archetype generation interactively

Properties available from archetype:

Name	Value

Add...

Remove

Advanced

< Back Next > Finish Cancel

- DIIDemo
 - src/main/java
 - src/test/java
 - JRE System Library [JavaSE-1.8]
 - Maven Dependencies
 - src
 - target
 - pom.xml

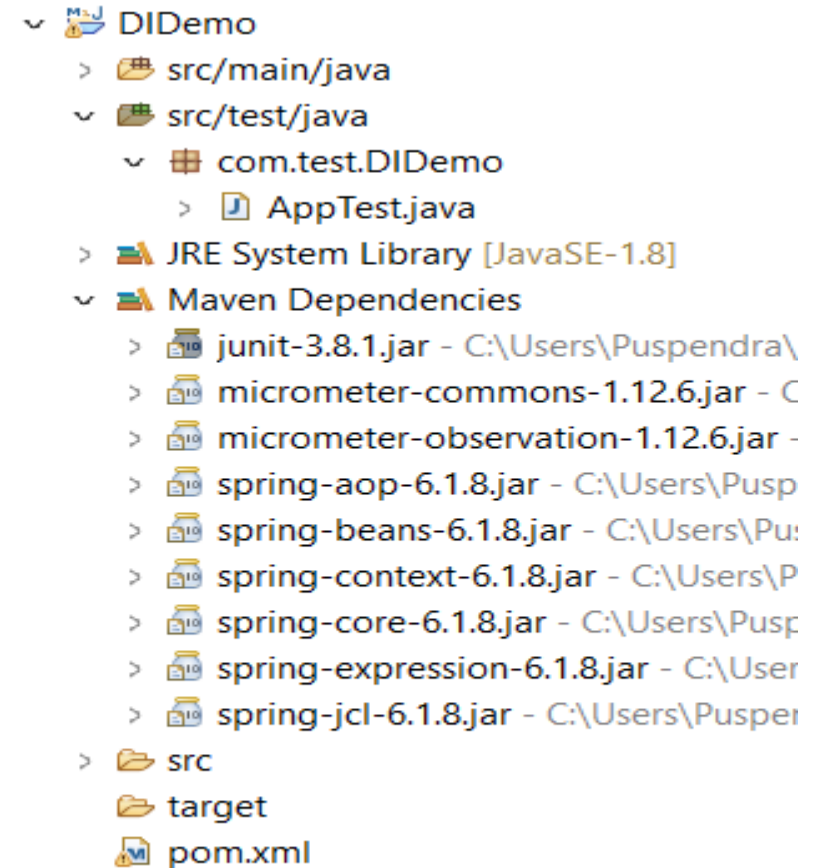
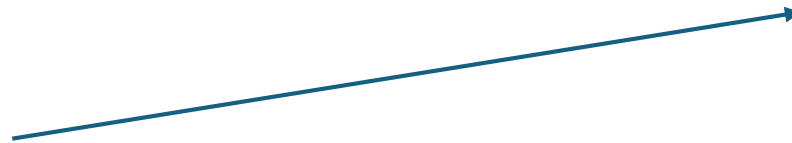
6. Add dependencies to <dependencies> tag

Press ctrl+s to save the pom.xml file(all dependencies will be downloaded automatically)

```
<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>spring-core</artifactId>  
<version>6.1.8</version>  
</dependency>
```

```
<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>spring-context</artifactId>  
<version>6.1.8</version>  
</dependency>
```

7. Check Project structure



8. Create new class Employee

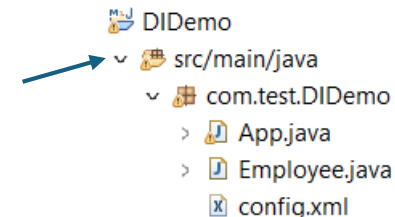
9. Create new xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
<bean class="com.test.DIDemo.Employee" name="stud1">
<property name="id" value="1" />
<property name="name" value="Raju" />
<property name="dept" value="Sales" />
</bean>
```

```
</beans>
```

```
package com.test.DIDemo;
public class Employee {
//POJO CLASS:FULLY ENCAPSULATED CLASS
private int empId;
private String name;
private String dept;
public Employee(int empId, String name, String dept) {
super();
this.empId = empId;
this.name = name;
this.dept = dept;
}
public int getEmpId() {
return empId;
}
public void setEmpId(int empId) {
this.empId = empId;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getDept() {
return dept;
}
public void setDept(String dept) {
this.dept = dept;
}
}
```

10. See project structure now




9. Update App.java

```
package com.test.DIDemo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        ApplicationContext context=new
        ClassPathXmlApplicationContext("com/test/DIDemo/config.xml");
        Employee employee=(Employee)context.getBean("stud1");
        System.out.println(employee);
    }
}
```

Dependency injection



10. execute App.java



```
Problems  @ Javadoc  Declaration  Console ×
<terminated> App (2) [Java Application] C:\Users\Puspendra\Downloads\eclipse-jee-2024-03
Hello World!
Employee [empId=1, name=Raju, dept=Sales]
```

Aspect-Oriented Programming (AOP)

- Aspect-Oriented Programming (AOP) is a programming technique that allows developers to modularize cross-cutting concerns. Cross-cutting concerns are tasks that affect multiple parts of a program, such as logging, security, and transaction management.
- AOP allows developers to separate these concerns from the main program logic. This makes the code more modular, reusable, and maintainable.
- Spring AOP is a popular implementation of AOP. It provides a simple and powerful way to write custom aspects.
- Spring provides simple and powerful ways of writing custom aspects by using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language while still using Spring AOP for weaving.
- AOP is used in the Spring Framework to:
- Provide declarative enterprise services. The most important such service is declarative transaction management.
- Let users implement custom aspects, complementing their use of OOP with AOP.

Benefits of using AOP

Modularity:

- AOP allows developers to separate cross-cutting concerns from the main program logic. This makes the code more modular, reusable, and maintainable.

Reusability:

- Aspects can be reused across multiple projects. This saves time and effort, and it can help to improve the consistency of code.

Maintainability:

- AOP makes it easier to maintain code. This is because cross-cutting concerns are separated from the main program logic. This makes it easier to understand and modify the code.

WebSocket API

Spring Framework provides a WebSocket API that adapts to various WebSocket engines, including Tomcat, Jetty, GlassFish, WebLogic, and Undertow. This API allows developers to easily implement WebSocket-based applications. The Spring Framework also provides a number of features that make it easy to develop WebSocket-based applications, including:

- A messaging framework that supports STOMP, a text-oriented messaging protocol that can be used over any reliable 2-way streaming network protocol such as TCP and WebSocket.
- A JavaScript client library that makes it easy to develop WebSocket-based web applications.
- A number of pre-built WebSocket-based applications, such as a chat application and a stock ticker.

BEAN SCOPE

- **Bean Scopes** refers to the lifecycle of Bean that means when the object of Bean will be instantiated, how long does that object live, and how many objects will be created for that bean throughout. Basically, it controls the instance creation of the bean and it is managed by the spring container.
- **In the Spring Framework, a bean's scope determines how long it lives and how many instances of it are created.**
- The default scope is **singleton**, Only one instance will be created for a single bean definition per Spring IoC container and the same object will be shared for each request made for that bean.
- The **prototype scope** A new instance will be created for a single bean definition every time a request is made for that bean. This is useful for beans that are not thread-safe or that need to be customized for each request.
- The **request scope** creates a new instance of the bean for each HTTP request. This is useful for beans that need to be associated with a specific request, such as a database connection or a shopping cart.
- The **session scope** creates a new instance of the bean for each user session. This is useful for beans that need to be associated with a specific user, such as a user profile or a shopping cart.
- The **global session scope** creates a new instance of the bean for each user session across all applications in the same cluster. This is useful for beans that need to be shared across multiple applications, such as a user profile or a shopping cart.

BEAN SCOPE

- You can specify the scope of a bean using the *@Scope* annotation. For example, the following code creates a bean with the prototype scope:

```
@Scope("prototype")  
public class MyBean {  
    // ...  
}
```

OR

```
<bean class="com.test.DIDemo.Employee" name="stud1"  
scope="request">  
    <property...  
</bean>
```

Autowiring

- **Autowiring** in the Spring framework can inject dependencies automatically.
- The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**.
- Autowiring in Spring internally uses constructor injection.
- An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

Modes	Description
No	This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.
byName	It uses the name of the bean for injecting dependencies.
byType	It injects the dependency according to the type of bean.
Constructor	It injects the required dependencies by invoking the constructor.
Autodetect(deprecated in Spring 3)	The autodetect mode uses two other modes for autowiring – constructor and byType.

1. No

This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

2. byName

It uses the name of the bean for injecting dependencies. However, it requires that the name of the property and bean must be the same. It invokes the setter method internally for autowiring.

```
<bean id="state" class="pack.State">  
  <property name="name" value="UP" />  
</bean>  
<bean id="city" class="pack.City" autowire="byName"></bean>
```

3. byType

It injects the dependency according to the type of the bean. It looks up in the configuration file for the class type of the property. If it finds a bean that matches, it injects the property. If not, the program throws an error. The names of the property and bean can be different in this case. It invokes the setter method internally for autowiring.

```
<bean id="state" class="sample.State">  
  <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City" autowire="byType"></bean>
```

4. constructor

It injects the required dependencies by invoking the constructor. It works similar to the “byType” mode but it looks for the class type of the constructor arguments. If none or more than one bean are detected, then it throws an error, otherwise, it autowires the “byType” on all constructor arguments.

```
<bean id="state" class="sample.State">
  <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="constructor"></bean>
```

5. autodetect

The autodetect mode uses two other modes for autowiring – constructor and byType. It first tries to autowire via the constructor mode and if it fails, it uses the byType mode for autowiring. It works in Spring 2.0 and 2.5 but is deprecated from Spring 3.0 onwards.

```
<bean id="state" class="sample.State">
  <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="autodetect"></bean>
```

Auto wiring example

```
class City {
private int id;
private String name;
private State s;
public City() {
}
public int getID() { return id; }
public void setId(int eid) { this.id = eid; }
public String getName() { return name; }
public void setName(String st) { this.name = st; }
public State getS() {
return s;
}
public void setS(State s) {
this.s = s;
}
public int getId() {
return id;
}
public City(State s) {
super();
this.s = s;
}
public City(int id, String name, State s) {
super();
this.id = id;
this.name = name;
this.s = s;
}
@Override
public String toString() {
return "City [id=" + id + ", name=" + name + ", s=" + s + "];"
}
}
```

dependency

```
package com.test.DemoProject;
import org.springframework.beans.factory.annotation.Autowired;
public class State {
private String name;
public String getName() { return name; }
public void setName(String s) { this.name = s; }
@Override
public String toString() {
return "State [name=" + name + "];"
}
public State(String name) {
super();
this.name = name;
}
public State() {
super();}
}
```

```
package com.test.DemoProject;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test {
public static void main(String[] args) {
ApplicationContext context = new
ClassPathXmlApplicationContext("/com/test/DemoProject/config1.xml");
City city=context.getBean("city", City.class);
System.out.println(city);
}
}
```

//main


```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/sche
ma/context"
xsi:schemaLocation="http://www.springframework.org
/schema/beans
http://www.springframework.org/schema/beans/spring
-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spri
ng-context.xsd">
<bean id="s" class="com.test.DemoProject.State" >
<property name="name" value="UP" />
</bean>
<bean name="city"
class="com.test.DemoProject.City"
autowire="constructor">
<property name="id" value="11" />
<property name="name" value="Washington, D.C." />
</bean>
</beans>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/schema
/context"
xsi:schemaLocation="http://www.springframework.org/s
chema/beans
http://www.springframework.org/schema/beans/spring-
beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring
-context.xsd">
<bean id="s" class="com.test.DemoProject.State" >
<property name="name" value="UP" />
</bean>
<bean name="city" class="com.test.DemoProject.City"
autowire="byName">
<property name="id" value="11" />
<property name="name" value="Washington, D.C." />
</bean>
</beans>
```

@autowired

There are three ways to apply the @Autowired annotation:

NOTE: PUT FOLLOWING LINE IN CONFIG.XML FILE

```
<context:annotation-config/>
```

1. On a field: This is the most common way to use the @Autowired annotation. Simply annotate the field with @Autowired and Spring will inject an instance of the dependency into the field when the bean is created.

```
public class MyBean {  
    @Autowired  
    private MyDependency dependency;  
}
```

2. On a constructor: You can also use the @Autowired annotation on a constructor. This will cause Spring to inject an instance of the dependency into the constructor when the bean is created.

```
public class MyBean {  
    private MyDependency dependency;  
    @Autowired  
    public MyBean(MyDependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

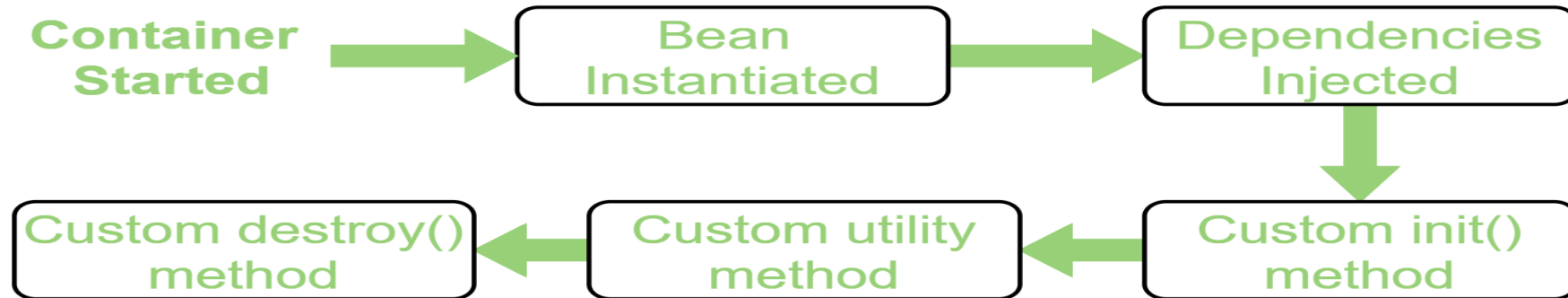
3. On a setter method: You can also use the `@Autowired` annotation on a setter method. This will cause Spring to inject an instance of the dependency into the setter method when the bean is created.

```
public class MyBean {  
    private MyDependency dependency;  
    @Autowired  
    public void setDependency(MyDependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

The `@Autowired` annotation can be used on any field, constructor, or setter method that is declared in a Spring bean. The dependency that is injected must be a Spring bean itself.

Life Cycle Call backs

- Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.



Example: Life Cycle Call Back

```
package beans;

public class HelloWorld {

    // This method executes
    // automatically as the bean
    // is instantiated
    public void init() throws Exception
    {
        System.out.println(
            "Bean HelloWorld has been "
            + "instantiated and I'm "
            + "the init() method");
    }

    // This method executes
    // when the spring container
    // is closed
    public void destroy() throws Exception
    {
        System.out.println(
            "Container has been closed "
            + "and I'm the destroy() method");
    }
}
```

```
<beans>
    <bean id="hw" class="beans.HelloWorld"
        init-method="init" destroy-
method="destroy"/>
</beans>

package test;

import beans.HelloWorld;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

// Driver class
public class Client {

    // Main driver method
    public static void main(String[] args) throws Exception
    {

        ConfigurableApplicationContext cap|
            = new ClassPathXmlApplicationContext(
                "resources/spring.xml");

        cap.close();
    }
}
```

Spring BOOT Using REST API

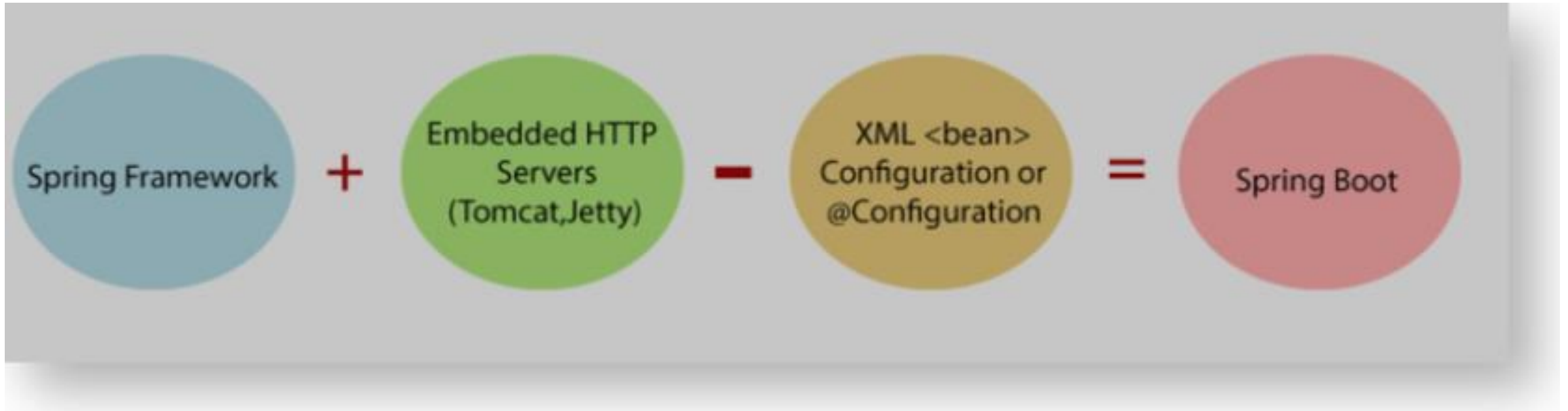
ABES Engineering College, Ghaziabad



Introduction

- Spring Boot is a project that is built on top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.
- It is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring Framework used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.

As summarized in the below figure, Spring Boot is the combination of Spring Framework and Embedded Servers.



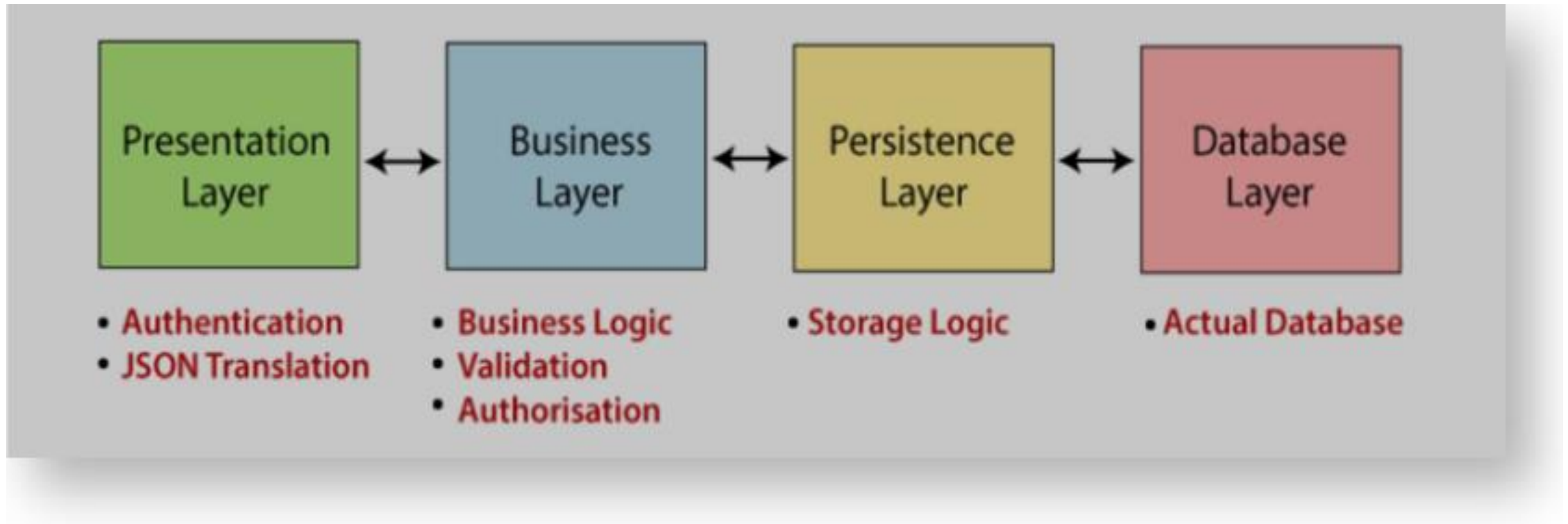
Spring Boot Architecture

- In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses [convention over configuration](#) software design paradigm which means that it decreases the effort of the developer.
- The main goal of Spring Boot is to reduce development, unit test, and integration test time and leveraging the following features:
- Create stand-alone Spring applications
- Embed Tomcat, Jetty, or Undertow directly (no need to deploy WAR files)

Spring Boot Architecture

- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration
- Provide opinionated 'starter' POMs to simplify your Maven configuration

Spring Boot Architecture



Spring Boot Architecture

- **Presentation Layer:** The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request, and transfer it to the business layer. In short, it consists of views.
Business Layer: The business layer handles all the business logic. It consists of service classes and uses services provided by data access layers. It also performs authorization and validation.
Persistence Layer: The persistence layer contains all the storage logic and translates business objects from and to database rows.
Database Layer: In the database layer, CRUD (create, retrieve, update, delete) operations are performed.

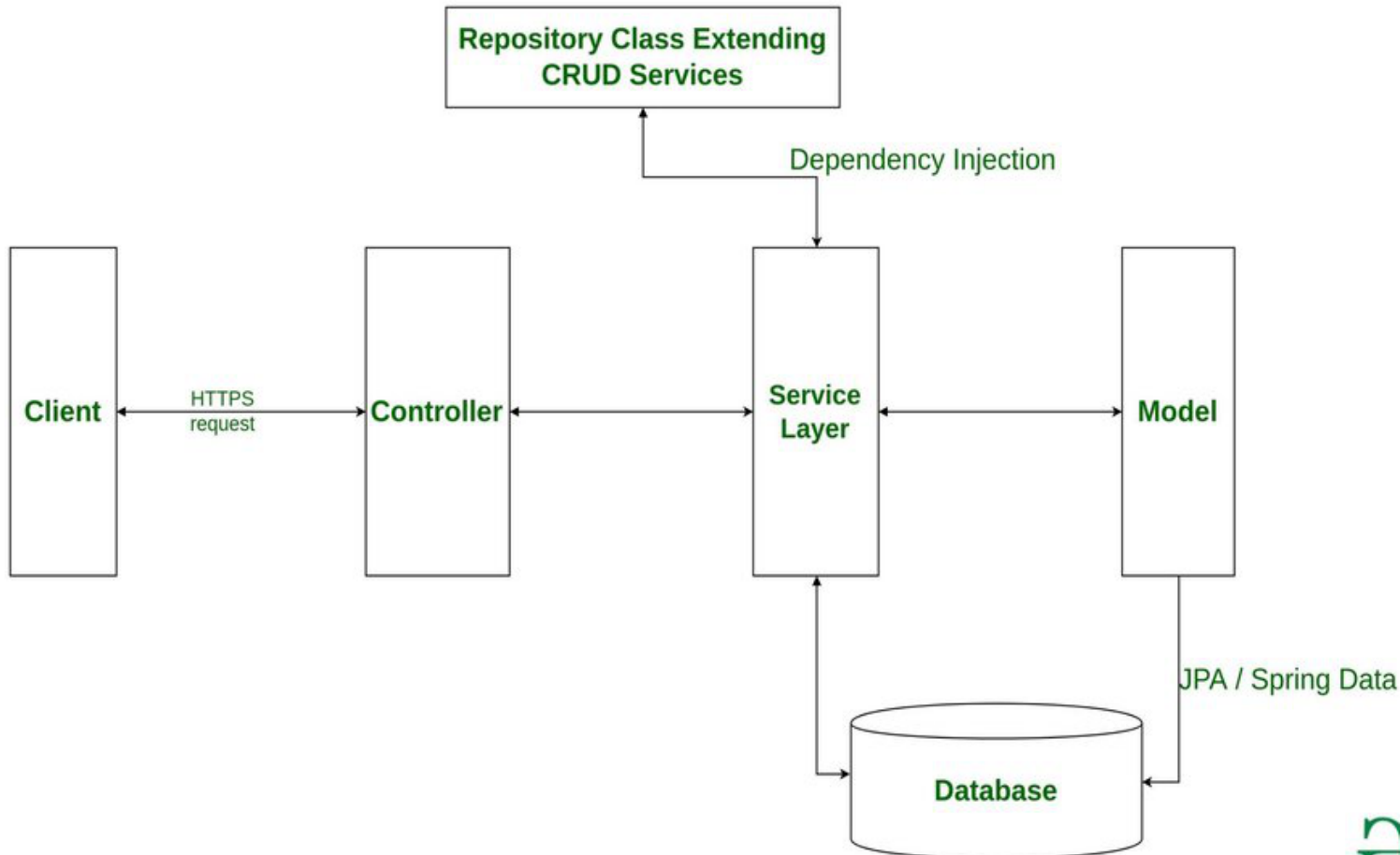
-

Spring Boot Architecture

- Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except for one thing: there is no need for DAO and DAOImpl classes in Spring boot. As a summary, in a simple spring boot flow:
- Data access layer gets created and CRUD operations are performed.
- The client makes the HTTP requests.
- The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
- In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
- A response page is returned to the user if no error occurs.
-

Spring Boot Architecture

Spring Boot flow architecture



- The Client makes an HTTP request(GET, PUT, POST, etc.)
- The HTTP request is forwarded to the Controller. The controller maps the request. It processes the handles and calls the server logic.
- The business logic is performed in the Service layer. The spring boot performs all the logic over the data of the database which is mapped to the spring boot model class through [Java Persistence Library](#)(JPA).
- The [JSP](#) page is returned as Response from the controller.

Spring Boot Starters

- Spring Boot starters are dependency descriptors
- Dependency: External libraries
- Descriptor: Configuration specific JARS and their Versions.
- Spring boot starter combines all the necessary libraries for a particular feature or technology into a single dependency.
- Example: web.jar, web-MVC.jar, validation.jar, tomcat.jar

Spring Tools 4 for Eclipse

The all-new Spring Tool Suite 4. Free.
Open source.

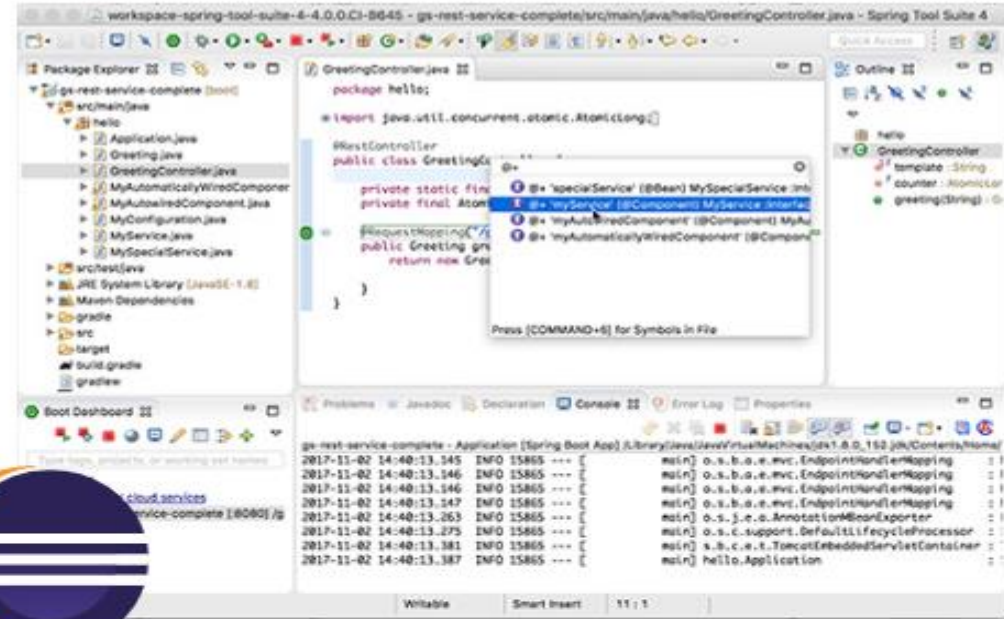
4.23.1 - LINUX X86_64

4.23.1 - LINUX ARM_64

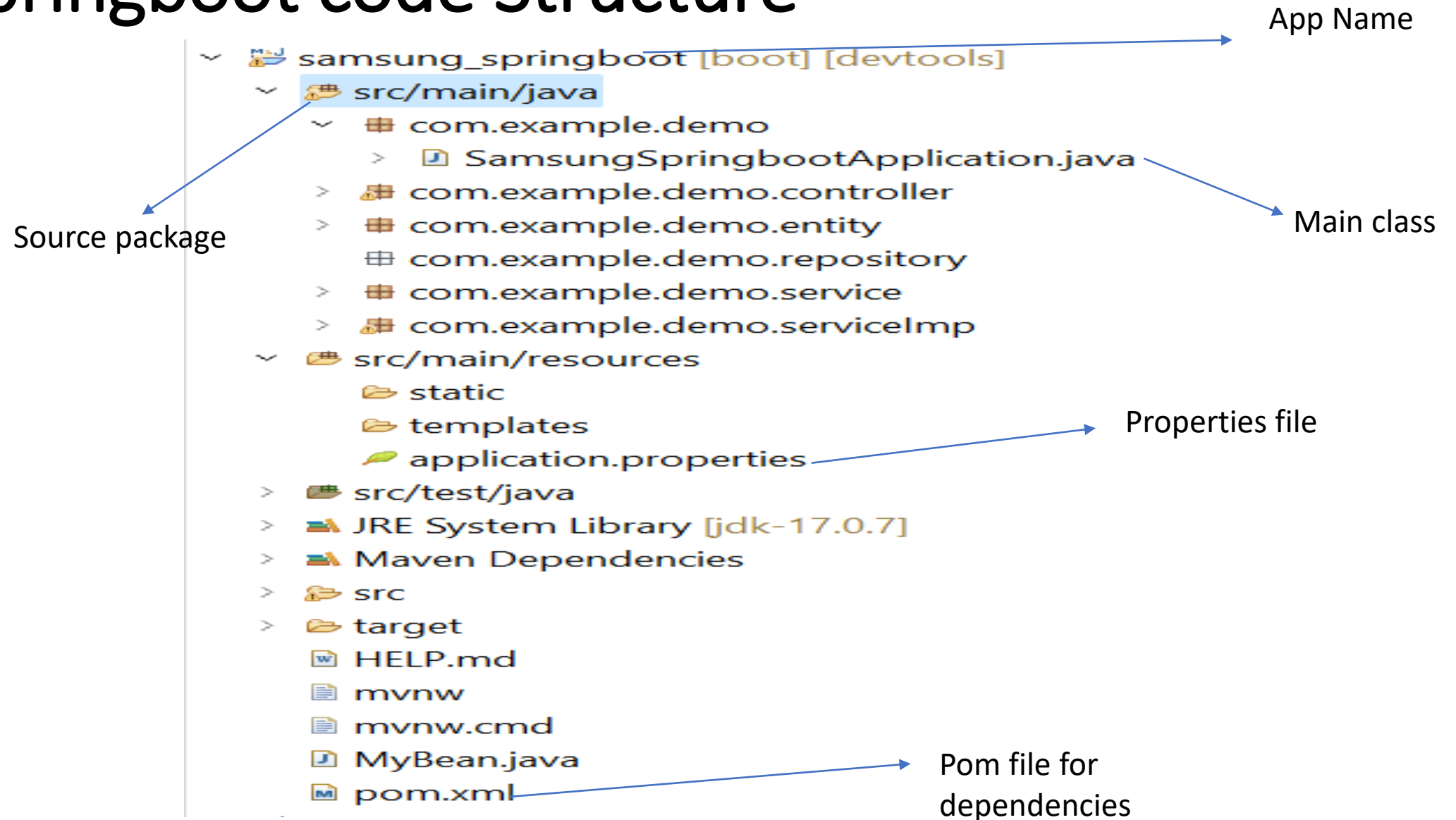
4.23.1 - MACOS X86_64

4.23.1 - MACOS ARM_64

4.23.1 - WINDOWS X86_64



Springboot code Structure



Spring Boot Annotations

- Spring Boot Annotations is a form of metadata that provides data about a program. Spring Boot Annotations are mostly placed in the following packages:
- `org.springframework.boot.autoconfigure`
- `org.springframework.boot.autoconfigure.condition`
-

Spring Boot Annotations

-

@Configuration: This annotation in Spring indicates that the class can be used to define bean configurations within the Spring application context

@ComponentScan: It tells Spring to scan the specified package and its sub-packages for components, configurations, and services to manage.

@EnableAutoConfiguration: It automatically configures the Spring application based on its dependencies and the contents of the classpath.

Spring Boot run() method

- Run Method Tasks :-
 - = Initialize Spring App: Sets up the Spring application.
 - = Configure Application Context: Establishes the application context.
 - = Load External Configuration: Loads external configurations.
 - = Load and Register Beans: Instantiates and registers beans.
 - = Apply Auto-Configuration: Automatically configures the application based on dependencies.
 - = Start Server (if Web): Initiates the server for web applications.

Spring Boot resources folder

- = The "static folder" is typically used to store static web resources such as HTML, CSS, JavaScript, images, and other assets.
- = These files are served directly by the web server without any processing by the backend application.
- = The "templates folder", is commonly used to store dynamic web pages or templates, which are typically processed by a templating engine like Thymeleaf, FreeMarker, or Velocity.
- = These templates are rendered dynamically by the server-side code before being sent to the client.
- = In "application.properties file", we typically provide configuration settings for various aspects of the Spring Boot application.
- = For example database settings, server port configuration, logging levels, and other application-specific properties.

Spring Boot Normal Application “pom.xml”

- <dependencies> *Add Spring Boot Starters...*
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter</artifactId>
 </dependency>

Spring Boot Web Application “pom.xml”

- = By providing only one starter i.e. “spring-boot-starter-web”, Spring Boot will automatically create Spring Boot Web Application
- = Automatically
 - > folder structure will be created
 - > jar files will be added
 - > configurations will be done

```
<dependencies>    Add Spring Boot Starters...  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-web</artifactId>  
    </dependency>
```

Spring Boot Runners

- In Spring Boot, runners are components used to execute code when the application is started. They are typically implemented using Spring's `ApplicationRunner` or `CommandLineRunner` interfaces. These runners allow you to perform tasks such as database initialization, data loading, or any custom startup logic.

Application Runner

- The ApplicationRunner interface in Spring Boot provides a way to execute code after the application context is initialized and before Spring Boot starts servicing incoming requests

```
1 package com.example.demo;
2
3 import org.springframework.boot.ApplicationArguments;
4
5
6
7
8 @SpringBootApplication
9 public class SamsungSpringbootApplication implements ApplicationRunner {
10
11     public static void main(String[] args) {
12         SpringApplication.run(SamsungSpringbootApplication.class, args);
13         System.out.println("Welcome to springboot appliation");
14     }
15
16     @Override
17     public void run(ApplicationArguments args) throws Exception {
18         // TODO Auto-generated method stub
19         System.out.println("ApplicationRunner is running using ApplicationRunner");
20     }
21
22 }
```

Problems Servers Properties Snippets Console Terminal

samsung_springboot - SamsungSpringbootApplication [Spring Boot App] C:\Program Files\Java\jdk-17.0.7\bin\javaw.exe (26-Jun-2024, 4

```
2024-06-26T16:52:10.812+05:30 WARN 25340 --- [ restartedMain] JpaBaseConfiguration
2024-06-26T16:52:11.138+05:30 INFO 25340 --- [ restartedMain] o.s.b.d.a.OptionalLi
2024-06-26T16:52:11.176+05:30 INFO 25340 --- [ restartedMain] o.s.b.w.embedded.tom
2024-06-26T16:52:11.183+05:30 INFO 25340 --- [ restartedMain] c.e.demo.SamsungSpri
ApplicationRunner is running using ApplicationRunner
Welcome to springboot appliation
```

CommandLine Runner

- Similar to ApplicationRunner, CommandLineRunner is an alternative interface that provides a run method with an array of String arguments. These arguments are passed directly to the application when it is started from the command line.

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class MyCommandLineRunner implements
CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        // Code to execute on application startup
        System.out.println("CommandLineRunner is running...");
    }
}
```

Springboot Logger

- In Spring Boot applications, logging is an essential aspect of monitoring and debugging. Spring Boot integrates with popular logging frameworks like Logback, Log4j2, and Java Util Logging (JUL). Here's a guide on how to use logging in a Spring Boot application with an example using Logback.

Steps to add Logger in Springboot App

Step1: Add dependency in POM.XML File

```
<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>2.0.13</version>
    </dependency>
```

- Step2: import relevant package in either Controller , Service or

```
2
3 @RestController
4 public class AddController {
5     private static final Logger logger = LoggerFactory.getLogger(AddController.class);
6
7     @Autowired
8     AddService addservice;
9     @Autowired
10    MessageService messageservice;
11
12    Student s1=new Student("Ram", 100, "IT");
13    Student s2=new Student("Ram", 100, "IT");
14    Student s3=new Student("Ram", 100, "IT");
15
16    @GetMapping("/addstudent")
17    public String addStudent() {
18        logger.debug("Debug message");
19        logger.info("Info message");
20        logger.warn("Warning message");
21        logger.error("Error message");
22        return addservice.addStudent(s1);
23    }
24 }
```

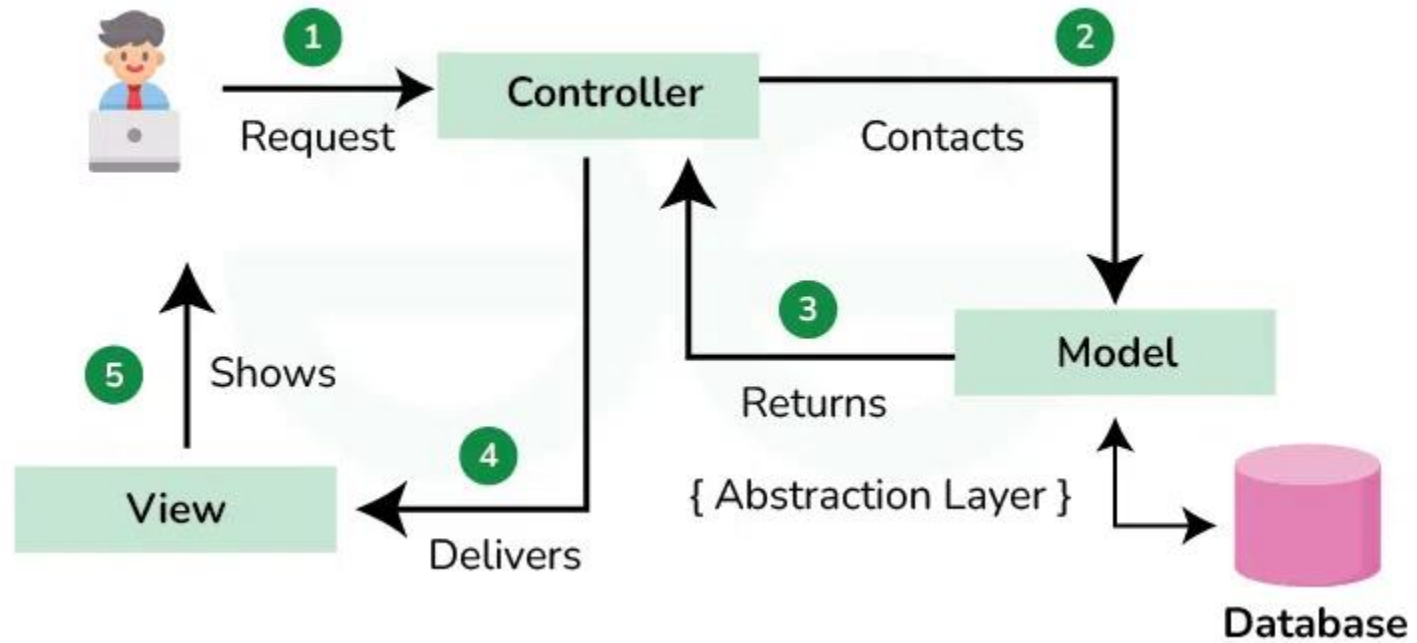
```
s.web.servlet.DispatcherServlet : initializing servlet dispatcherServlet
s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
example.demo.controller.AddController : Info message
example.demo.controller.AddController : Warning message
example.demo.controller.AddController : Error message
```

O/P

- **Example Use Cases**

- **Database Initialization:** Load initial data or perform schema setup.
- **External Service Initialization:** Connect to external services or APIs.
- **Cache Warming:** Preload caches on application startup.
- **Logging:** Output informational messages or logs about application state.

- MVC Design Pattern



Restful Web Services

- REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding**, who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

Restful Web Services

- It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:
- The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource.

Restful Web Services

- The important methods of HTTP are:
- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

Restful Web Services

- Advantages of RESTful web services
- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like **JSON, text, HTML, and XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- They are **language neutral**.
-

Spring Boot Annotations

1. Spring – REST Controller

- **RestController:** RestController is used for making restful web services with the help of the `@RestController` annotation. This annotation is used at the class level and allows the class to handle the requests made by the client. Let's understand `@RestController` annotation using an example. The RestController allows to handle all REST APIs such as [GET](#), [POST](#), [Delete](#), [PUT](#) requests.

Create a Rest Controller

Create a new package as `com.example.demo.controller` and create a new class as **SampleController** , now make this class as RestController using **@RestController** annotation

```
1 package com.example.demo.controller;
2
3 import org.springframework.web.bind.annotation.RestController;
4
5 @RestController
6 public class SampleController {
7
8 }
9
```

Now, Apply @GetMapping annotation to create an end point to access this controller, given below is code:

```
6 @RestController
7 public class SampleController {
8
9     @GetMapping("/samplemsg")
10    public String hi() {
11
12        return "Sample Controller message";
13    }
14
15 }
16
```

2. PostMapping

```
    ,  
  
    @PostMapping("/addition")  
    public long add() {  
        return 23+67;  
    }
```

3. Spring – Request Body

- `@RequestBody`: Annotation is used to get the request body in the incoming request.

4. Spring – Request Mapping and ResponseBody

- **@RequestMapping** Annotation which is used to map HTTP requests to handler methods of MVC and REST controllers.
- The **@RequestMapping** annotation can be applied to class-level and/or method-level in a controller.
- The class-level annotation maps a specific request path or pattern onto a controller.
- You can then apply additional method-level annotations to make mappings more specific to handler methods.
- **@ResponseBody** annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the `HttpResponse` object. When you use the **@ResponseBody** annotation on a method, Spring converts the return value and writes it to the HTTP response automatically.

5. Spring – Request Mapping and ResponseBody

```
1 package com.example.demo.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestMethod;
6 import org.springframework.web.bind.annotation.ResponseBody;
7
8 @Controller
9 public class SampleController {
10
11     @RequestMapping(value = "/hello", method = RequestMethod.GET)
12     @ResponseBody
13     public String hello() {
14         return "Hello, World!";
15     }
16 }
17
```

6. Spring – PathVariable

- The **@PathVariable** annotation is used to extract the value from the URI. It is most suitable for the RESTful web service where the URL contains some value. Spring MVC allows us to use multiple @PathVariable annotations in the same method. A path variable is a critical part of creating rest resources.

```
@GetMapping(path="/hello-world/path-variable/{name}")  
public HelloWorldBean helloWorldPathVariable(@PathVariable String name)  
{  
return new HelloWorldBean(String.format("Hello World, %s", name));  
}
```

URL: <http://localhost:8080/hello-world/path-variable/ABES>
ABES”}

O/P- {“message”:”Hello World,

7. Spring – Request Parameter

- The `@RequestParam` annotation is used to extract data from the query parameters in the request URL. Query parameters are the key-value pairs that appear after the `?` in a URL.

```
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.ResponseBody;
7
8 @Controller
9 public class SampleController {
10
11     @GetMapping("/hello")
12     @ResponseBody
13     public String hello(@RequestParam String name) {
14         return "Hello, " + name + "!";
15     }
16 }
17
```

Rest API

- **REpresentational State Transfer (REST)** is a software architectural style that developers apply to web application programming interfaces ([APIs](#)).
- REST APIs are the most common APIs used across the web today because the REST pattern provides simple, uniform interfaces. These can be used to make data, content, algorithms, media, and other digital resources available through web URLs, so that they can be consumed within web, mobile, and device applications.

Rest API

- let's consider my own personal Facebook presence, where I am a resource. I can view an HTML representation of my resource at:
- GET <https://www.facebook.com/prashant.tomer.946/>
- My profile is a single resource available on Facebook. I can view a representation of that resource in HTML using that URL in my web browser. I can also view another representation of it using the Facebook Graph API.
- GET <https://graph.facebook.com/v7.0/me>

Rest API

- This is just one resource Facebook provides, offering up a buffet of digital resources for me to consume as a developer.
- **Comments:** GET <https://graph.facebook.com/v7.0/comments>
- **Friends:** GET <https://graph.facebook.com/v7.0/friends>
- **Images:** GET <https://graph.facebook.com/v7.0/images>
- **Links:** GET <https://graph.facebook.com/v7.0/links>
- **Likes:** GET <https://graph.facebook.com/v7.0/likes>

Rest API

- This allows for a handful of operations on each resource using HTTP methods:
- **GET:** <https://graph.facebook.com/v7.0/images>
- **POST:** <https://graph.facebook.com/v7.0/images>
- **PUT:** <https://graph.facebook.com/v7.0/images>
- **DELETE:** <https://graph.facebook.com/v7.0/images>

Types of HTTP Methods and Mapping in Spring framework

HTTP Method	Mapping Method Annotation in Spring Boot
Get Method	@GetMapping
Post Method	@PostMapping
Delete Method	@DeleteMapping
Put Method	@PutMapping
Patch Mapping	@PatchMapping

Create Spring boot App using Rest API

- **Tools Required in this Spring Boot Application:**
 - **Eclipse IDE: For project development**
 - **Postman Application: Rest API Testing**
 - **Spring Boot API: For Spring boot libraries**
 - **JDK 8: base of java core development**

Steps to Create SpringBoot project and use RestAPI
























Step1: Download springboot skeleton from spring initializer.

Step 2: import **step1** project in eclipse IDE.

Step3: Open main file where you get main method and run that class as usual do in eclipse to run the normal class.

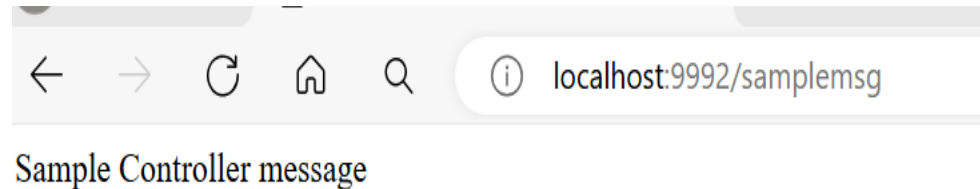
➤ Step4: Observe the output.

Project Skeleton

- ✓  samsung_springboot [boot] [devtools]
 - ✓  src/main/java
 - ✓  com.example.demo
 - >  SamsungSpringbootApplication.java
 - >  com.example.demo.controller
 - >  com.example.demo.entity
 -  com.example.demo.repository
 - >  com.example.demo.service
 - >  com.example.demo.serviceImp
 - ✓  src/main/resources
 -  static
 -  templates
 -  application.properties
 - >  src/test/java
 - >  JRE System Library [jdk-17.0.7]
 - >  Maven Dependencies
 - >  src
 - >  target
 -  HELP.md
 -  mvnw
 -  mvnw.cmd
 -  MyBean.java
 -  pom.xml

Run Application

- To test your REST end point, need to run your springboot application and check on which port your application is running (check console and analyse the output , port number is also there), now go to browser and access using your end point name:



Thank You