**What is Dependency injection? Why it is needed?**

In a normal life if we are a Laptop manufacturer we need to add different parts of Laptop from different company like Screen can be from Samsung, HardDrive can be from Hitachi, Keyboard can be of Lenovo so these all are depenedent to manufacture a laptop. So these all are dependency

Similarly in Spring framework if we have a java class that can be dependent on other class object

Like:

Class Laptop {

HardDrive hd;

Monitor m;

Keyboard key;

}

But when we need to use these all objects we need to create the object of different parts like: HardDrive h = new HitachiHardDrive() ---🡪Tight coupling

Using Dependency injection we can make it loosely coupled.

Interface HardDrive { }

Class HitachiHardDrive implements HardDrive { }

On using this we need to call HardDrive hd = new HitachiHardDrive();

In future if we need to change HitachiHardDrive with SamsungHardDrive ,

We need to change only the object.

**But how these dependency injection occurs?**

We need to do it either by using xml or by using **@Componet**

**Spring container** is responsible to create the object and provide it to Laptop class by using **@Autowire**

**Class Laptop{**

**@Autowired**

**HardDrive hd;**

**}**

**@Component**

Class HitachiHardDrive implements HardDrive{ -----}

By using these two annotations Spring container creates the object and provide it to Laptop

2nd Use of DI :

For testing our application we need to mock an object using DI

**What is Different scope Singleton and Prototype in spring Boot?**

**Ans :** Using Singleton scope :🡪

Only one object of bean is being created on calling getBean(Employee.class) .

if we don’t call getBean(Employee.class) , a single object is being created by Spring container

Using prototype scope :🡪

Each time a new object a created whenever we call .getBean(abc.class)

If we don’t call this method there will not be any object is being created by Spring container

public class Employee {  
 private int id;

private String name;  
 private float salary;  
  
 public Employee() {  
 System.*out*.println("Bean is created");  
 }  
  
 public void show(){  
 System.*out*.println("HIII");  
  
 }  
}

@SpringBootApplication  
public class ScopeAndDiApplication {  
  
 public static void main(String[] args) {  
  
 ConfigurableApplicationContext context = SpringApplication.*run*(ScopeAndDiApplication.class, args);  
 Employee e1 = context.getBean(Employee.class);  
 e1.show();  
 }  
}

In this case it will show :

Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:

**No qualifying bean of type 'com.example.scopeAndDI.Employee' available**

So We need to add **@Component** with Employee class

**O/P : Bean is created**

**HIII**

Now if We create two beans of Employee :

Employee e1 = context.getBean(Employee.class);  
e1.show();  
Employee e2 = context.getBean(Employee.class);  
e2.show();

**O/P : Bean is created**

**HIII**

**HIII**

Only one Bean is created because by default scope is **Singleton**

If we put **@Scope(value = “prototype”)**

**O/P :**

**Bean is created**

**HIII**

**Bean is created**

**HIII**

If we don’t create a bean i.e we comment

// Employee e1 = context.getBean(Employee.class);

No any bean is created in this case but in case of Singleton scope after commenting also a new bean were created by Spring container.

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

@Autowired :🡪By default Autowired search by type

@Qualifier 🡪 Be default Qualifier searches by name

@Component("add")  
public class Address {  
 private String city;  
 private String state;  
 private String country;  
  
}

@Component  
@Scope(value="prototype")  
public class Employee {  
 private int id;  
 private String name;  
 private float salary;  
 @Autowired  
 @Qualifier("add")  
 private Address address;

* Spring boot normally doesn’t support jsp page i.e does not return jsp page i.e it is mainly intended for rest api not jsp

@Controller  
public class HomeController {  
 @RequestMapping("home")  
 public String home(){  
 return "home.jsp";  
 }  
}

On hitting https://localhost:8080/home

This code doesnot send us to home.jsp

* we need to add the dependency of tomcat-jasper to redirect it to that jsp

and automatically Spring boot searches the home.jsp under **src/main/webapp**

**Scenario:** If we do any configuration change manually we need to change that configuration in **application.properties** file

If we want some changes on this Project like I want to move home.jsp page in another folder **src/main/webapp.pages** and in controller under home method we **remove the file extension .jsp** because in future it may change to **.velocity or .html**

@Controller  
public class HomeController {  
 @RequestMapping("home")  
 public String home(){  
 return "home";  
 }  
}

**Solution:**

We use **prefix for path location and suffix for file extension** under properties file

spring.mvc.view.prefix = /pages/

spring.mvc.view.suffix = .jsp

similarly we can configure all the database configuration etc

Accepting Client data:

If we are pssing some dynamic data from client side using url like : [**https://localhost:8080/home?name=deepak**](https://localhost:8080/home?name=deepak)

This will be accepted by **Controller (server)** using **HttpServletRequest** object and it is forwarded to jsp using **HttpSession**

@Controller  
public class HomeController {  
 @RequestMapping("home")  
 public String home(HttpServletRequest request){  
 HttpSession session= request.getSession();  
 String name = request.getParameter("name");  
 session.setAttribute("username",name);  
 return "home";  
 }  
}

And jsp can take this data using jstl like :

</head>  
<body>  
 Welcome ${username}  
</body>  
</html>

Spring Boot using ModelAndView :

The above way to Accepting client data to server and forwarded to jsp is servlet specific so We can do it more effectively with simple manner using ModelAndView abject

@Controller  
public class HomeController {  
 @RequestMapping("home")  
 public ModelAndView home(@RequestParam("name") String myName){  
 ModelAndView modelAndView = new ModelAndView();  
 modelAndView.addObject("name",myName);  
 modelAndView.setViewName("home");  
 return modelAndView;  
 }  
}

In this way we can add so many objects also.

To finish our application, we need to create a single Java file. By default, Maven compiles sources from src/main/java, so you need to create that folder structure and then add a file named src/main/

java/Example.java to contain the following code:

**import** org.springframework.boot.\*;

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.web.bind.annotation.\*;

@RestController

@EnableAutoConfiguration

**public class** Example {

@RequestMapping("/")

String home() {

**return *"Hello World!"***;

}

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

SpringApplication.run(Example.**class**, args);

}

**The @RestController and @RequestMapping Annotations**

The first annotation on our Example class is @RestController. This is known as a *stereotype*

annotation. It provides hints for people reading the code and for Spring that the class plays a specific

role. In this case, our class is a web @Controller, so Spring considers it when handling incoming

web requests.

The @RequestMapping annotation provides “routing” information. It tells Spring that any HTTP request

with the / path should be mapped to the home method. The @RestController annotation tells Spring

to render the resulting string directly back to the caller.

**The @EnableAutoConfiguration Annotation**

The second class-level annotation is @EnableAutoConfiguration. This annotation tells Spring Boot

to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since

spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration assumes that

you are developing a web application and sets up Spring accordingly.

**The “main” Method**

The final part of our application is the main method. This is just a standard method that follows

the Java convention for an application entry point. Our main method delegates to Spring Boot’s

SpringApplication class by calling run. SpringApplication bootstraps our application, starting

Spring, which, in turn, starts the auto-configured Tomcat web server. We need to pass Example.class

as an argument to the run method to tell SpringApplication which is the primary Spring component.

The args array is also passed through to expose any command-line arguments.

**11.4 Running the Example**

At this point, your application should work. Since you used the spring-boot-starter-parent POM,

you have a useful run goal that you can use to start the application. Type mvn spring-boot:run