Table of Contents

Step 1: Understand Spring Boot

1. View the pom.xml file that has spring boot starter parent.

   It is a special starter project that provides default configurations for our application and a complete dependency tree to quickly build our Spring Boot project.
   It also provides default configurations for Maven plugins, such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, and maven-war-plugin.
   Beyond that, it also inherits dependency management from spring-boot-dependencies, which is the parent to the spring-boot-starter-parent.

2. @SpringBootApplication on the class with the main method:

   This annotation is used to enable three features, that is:
   a. @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
   b. @ComponentScan: enable @Component scan on the package where the application is located (see the best practices)

   c. @Configuration: allow to register extra beans in the context or import additional configuration classes

3. Run the application, you should see the error with respect to database configuration.
   This is the default behaviour of spring boot. It sees mysql dependency but did not find database connection parameter information.

4. Open application.properties file within the resources folder and add the database related connection parameters:

```
# windows users use 3306 as port number
#spring.datasource.url=jdbc:mysql://localhost:3306/java
spring.datasource.url=jdbc:mysql://localhost:8889/java
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

# below property specifies the dialect to use
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect

spring.sql.init.platform=mysql

# below property if used will perform DDL operations to create table
spring.jpa.generate-ddl=true

# below property if used will show the sql queries generated by hibernate
spring.jpa.show-sql=true
```

5. Following the package structure is very important with spring boot for it to follow the default configurations.

Step 2: Understand DI in spring boot

BELOW CODE IS TO BE ADDED IN project created

Create below classes within package com.demo.service:

6. Create an interface IMessageProvider as follows:

```
public interface IMessageProvider {
   public String getMessage();
}
```

7. Create ScannerMessageProvider as follows:

```
public class ScannerMessageProvider implements  IMessageProvider {
   @Override
   public String getMessage()
   {
      return "From Scanner";
   }
}
```

8. Create StringMessageProvider as follows:

```
public class SringMessageProvider implements  IMessageProvider {

    private String message;
    public void setMessage(String message)
    {
            this.message = message;
     }
   @Override
   public String getMessage()
   {
      return message;
    }
}
```

9. Create a class NotificationService as follows:

```
public class NotificationService {

     IMessageProvider provider;

    public NotificationService(IMessageProvider provider)
    {
            this.provider = provider;
    }

    public void setMessage(IMessageProvider provider)
     {
            this.provider = provider;
      }

    public void sendMessage()
    {
       System.out.println(provider.getMessage());
    }
}
```

10. Add @Component annotation as follows, that informs spring to create object of the class

```
@Component
public class SringMessageProvider implements  IMessageProvider {
            …..
}
```

Execute the application:

1. Now run the application and you will notice the beans created as output from constructor will be printed

2. To get the reference of spring context update the main method as follows:

   ```
   ApplicationContext context = SpringApplication.run(SpringBootDemo.class, args);
   ```

3. Now once you have the context, get the bean reference using context.getBean and will be able to call respective methods

4. Complete the class which has main method as follows for spring to set the context and scan for all the classes within the same or sub packages for spring specific annotations. If spring creates objects of the class they are called as “SPRING MANAGED BEAN”

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

        // NOT A SPRING MANAGeD BEAN
        // StringMessageProvider messageProvider = new StringMessageProvider();
        System.out.println();
        for(String beanName: context.getBeanDefinitionNames())
            System.out.println(beanName);
    }
}
```

@SpringBootApplication on the class with the main method:

This annotation is used to enable three features, that is:

@EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
@ComponentScan: enable @Component scan on the package where the application is located (see the best practices)
@Configuration: allow to register extra beans in the context or import additional configuration classes

Step 3: Get bean context

1. To get the reference of beans created by spring, we can specify classname:

   ```
   StringMessageProvider messageProvider = context.getBean(StringMessageProvider.class);
   ```

2. To get the reference of beans created by spring, we can specify id as well which is auto-generated by spring using camel casing syntax as follows:

   NOTE: typecasting is required

   ```
   StringMessageProvider messageProvider = (StringMessageProvider
                                                  )context.getBean("stringMessageProvider");
   ```

3. You can give your id as well as follows:

   ```
   @Component("ob")
   public class SringMessageProvider implements  IMessageProvider {
           …..
   }
   ```

   ```
   StringMessageProvider messageProvider = (StringMessageProvider )context.getBean("ob");
   ```

4. Update App class as follows:

   ```
   StringMessageProvider messageProvider = (StringMessageProvider)
   context.getBean("stringMessageProvider");
   messageProvider.setMessage("Hey!!");
   System.out.println(messageProvider.getMessage());
   ```

Step 4: @Value Annotation

1. Instead of manually setting the value for message, it can be injected using @Value annotation. @Value is used for primitive data types only. Update StringMessageProvider as follows:

   ```
   @Component
   public class SringMessageProvider implements  IMessageProvider {

       @Value("Hey injected by spring")
       private String message;
       public void setMessage(String message)
       {
               this.message = message;
         }
      @Override
      public String getMessage()
      {
         return message;
      }
   }
   ```

2. Update App class as follows: Message should be from @BValue

   ```
   StringMessageProvider messageProvider = (StringMessageProvider) context.getBean("stringMessageProvider");
   System.out.println(messageProvider.getMessage());
   ```

Step 5: Scoping In Spring

1. Default spring scope is singleton. Add below code in App class main method and you will see that no matter how many times you call getBean, it is always the same refernce

```
StringMessageProvider messageProvider1 = context.getBean(StringMessageProvider.class);

StringMessageProvider messageProvider2 = context.getBean(StringMessageProvider.class);

System.out.println(messageProvider1.getMessage());
System.out.println(messageProvider2.getMessage());

messageProvider1.setMessage("Hey!!");
System.out.println(messageProvider1.getMessage());
 System.out.println(messageProvider2.getMessage());
```

2. To change the scope to prototype add below on the String provider as class and then changes in messageProvider1 will not reflect in messageProvider2

```
@Component
@Scope("prototype")
public class SringMessageProvider implements  IMessageProvider {
….
}
```

Step 6: @Autowired Annotation

1. For spring to inject value for primitive types use @Value annotation as follows

```
public interface IMessageProvider {
   public String getMessage();
}
```

2. To inform spring to instantiate the respective classes add @Component as follows above the 2 classes :

```
@Component
public class NotificationService{
……
}

@Component
public class StringMessageProvider {
……
}
```

3. To inject the dependency of Message Provider in Notification service we use @Autowired annotation 3 ways.

   a. Constructor injection
   b. Setter injection
   c. Field injection

   NOTE: We do DI either on the one location and not all. But for practice you can try on 3 by commenting and uncommenting

```
//@Autowired
IMessageProvider provider;

@Autowired
public NotificationService(IMessageProvider provider)
{
      this.provider = provider;
}
//@Autowired
public void setMessage(IMessageProvider provider)
 {
            this.provider = provider;
}
```

4. Update the main method as follows:

// Comment out the code for String Message Provider

```
NotificationService notificationService =  context.getBean(NotificationService.class);
System.out.println(notificationService.sendMessage());
```

Step 7: @Qualifier Annotation

1. Add @Component on ScannerMessageProvider as well.
2. Now running the main method, gives an error as 2 beans are available for dependency injection String and Scanner Message provider.
3. To resolve this issue, use @Qualifier in NotificationService class as follows depending on which class object you need spring to inject

```
@Autowired
@Qualifier("stringMessageProvider") // "scannerMessageProvider"
IMessageProvider provider;
```

Step 8: @Bean Annotation

1. CREATE a simple maven java project “PaymentProject” with quickstart artifact and create a class as follows:

Below code is within PaymentProject which is a Java Maven Project.
[PLEASE NOTE : IT IS NOT SPRING PROJECT]

It has only one class as follows

```
package com.payment;

public class PaymentService {

   public double makePayment(double amount, double discount){
      amount = amount - amount * discount/100;
      double total = amount + 0.18;
      return  total;
   }
}
```

VVIMP:
Run maven install to create a jar file of PaymentProject and install within local repository.

2. Add PaymentProject as maven dependency in pom.xml of **SpringCoreDemo project**

```
<dependency>
          <groupId>com.payment</groupId>
          <artifactId>PaymentProject</artifactId>
          <version>1.0-SNAPSHOT</version>
</dependency>
```

3. Create a class BillingService within service package in previous **SpringCoreDemo** project as follows

```
@Component
public class BillingService {

   /**
    * payment service is not a part of this project and has been added as a dependency.
    * IN this case we cannot add @Component on the PaymentService class.
    */

   @Autowired
   private PaymentService paymentService;


   public void callService()
   {
      System.out.println(paymentService.makePayment(12000,10));
    }
}
```

4. To inject PaymentService as a dependency, use @bean annotation as follows within the App class :

```
@Bean
    public PaymentService paymentService()
    {
        return new PaymentService();
    }
```

5. Update the App class main method as follows:

```
BillingService bservice = context.getBean(BillingService.class);
bservice.callService();
```

Step 9: Collection Dependency Injection and @Configuration annotation

1. CREATE class as follows to see a demo on collection

```
public class Author {
    private String id;
    private String name;
    public Author(){
        System.out.println("Author def");
    }

    public Author(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Author{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                '}';
    }
}
@Component
public class CollectionDemo {

@Autowired
private List<String> fruits;

@Autowired
private List<Author> authors;

@Autowired
private Set<Integer> ids;

@Autowired
private Map<String, Integer> map;

public List<String> getFruits() {
return fruits;
}




public void setFruits(List<String> fruits) {
this.fruits = fruits;
}

public List<Author> getAuthors() {
return authors;
}

public void setAuthors(List<Author> authors) {
this.authors = authors;
}

public Set<Integer> getIds() {
return ids;
}

public void setIds(Set<Integer> ids) {
this.ids = ids;
}

public Map<String, Integer> getMap() {
return map;
}

public void setMap(Map<String, Integer> map) {
this.map = map;
}

}
```

2. Create a class AppConfig and add below code in the same

```
public class AppConfig{
    @Bean
    public List<String> getFruits()
    {
    System.out.println("fruits created");
    return Arrays.asList("Apples","Oranges","Grapes");
    }

    @Bean
    public Set<Integer> getIds()
    {
    System.out.println("ids created");
    Set<Integer> ids = new HashSet<Integer>();
    ids.add(1);
    ids.add(2);
    ids.add(3);
    return ids;
    }
    @Bean
    public List<Author> initializeData()
    {
    System.out.println("authors created");
    return Arrays.asList(new Author("A101", "AA"),new Author("A102", "BB"));
```

```
        }

        @Bean
        public Map<String, Integer> getMap()
        {
        System.out.println("map created");
        Map<String, Integer> map = new HashMap<String, Integer>();
        map.put("S1", 1);
        map.put("S2", 2);
        return map;
        }
}
```

3. Now add below code in main class:

```
CollectionDemo demo = context.getBean(CollectionDemo.class);
System.out.println(demo.getFruits());
System.out.println(demo.getAuthors());
System.out.println(demo.getIds());
System.out.println(demo.getMap());
```

   It will throw an error as none of the methods with @Bean were invoked. To call these methods, add @Configuration annotation on the AppConfig class as follows:

```
@Configuration
public class AppConfig{
    ……
}
```

   Now running main method works as expected

Step 10: @Lazy

1. CREATE 2 classes as follows to see a demo on @Lazy annotation

```
@Component
@Lazy
public class LazyService {
    public LazyService() {
        System.out.println("Lazy service intsantiated");
    }

}

@Component
public class EagerService {
    public LazyService() {
        System.out.println("Eager service intsantiated");
    }

}
```

2. Now in App class, when spring is loading the beans, Lazy service will not be loaded by default unless you ask for that bean using context.getBean(LazyService.class)

Step 11: @Primary

1. Since there was more than 1 implementation of IMessageProvider, we can use @Primary annotation on one of the class to make it a primary dependency.

```
@Primary
public class StringMessageProvider implements IMessageProvider {
….
}
```

2. Now string mesagae provider will be the default dependency injected by spring within NotificationService but we can still use @Qualifier to override the default