Spring Boot

1) What is Spring Boot?

Spring Boot is a Spring module which provides RAD (Rapid Application Development) feature to Spring framework.

It is used to create stand alone spring based application that you can just run because it needs very little spring configuration.

For more information click here.

2) What are the advantages of Spring Boot?

- Create stand-alone Spring applications that can be started using java -jar.
- Embed Tomcat, Jetty or Undertow directly. You don't need to deploy WAR files.
- It provides opinionated 'starter' POMs to simplify your Maven configuration.
- It automatically configure Spring whenever possible.

For more information click here.

3) What are the features of Spring Boot?

- Web Development
- SpringApplication
- Application events and listeners
- Admin features

For more information click here.

4) How to create Spring Boot application using Maven?

There are multiple approaches to create Spring Boot project. We can use any of the following approach to create application.

- Spring Maven Project
- Spring Starter Project Wizard
- Spring Initializr
- Spring Boot CLI

For more information click here.

5) How to create Spring Boot project using Spring Initializer?

It is a web tool which is provided by Spring on official site. You can create Spring Boot project by providing project details.

For more information click here.

6) How to create Spring Boot project using boot CLI?

It is a tool which you can download from the official site of Spring Framework. Here, we are explaining steps.

Download the CLI tool from official site and For more information click here.

7) How to create simple Spring Boot application?

To create an application. We are using STS (Spring Tool Suite) IDE and it includes the various steps that are explaining in steps.

For more information click here.

8) What are the Spring Boot Annotations?

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class. We need to import org.springframework.web.bind.annotation package in our file, in order to implement it.

For more information click here.

9) What is Spring Boot dependency management?

Spring Boot manages dependencies and configuration automatically. You don't need to specify version for any of that dependencies.

Spring Boot upgrades all dependencies automatically when you upgrade Spring Boot.

For more information click here.

10) What are the Spring Boot properties?

Spring Boot provides various properties which can be specified inside our project's **application.properties** file. These properties have default values and you can set that inside the properties file. Properties are used to set values like: server-port number, database connection configuration etc.

For more information click here.

11) What are the Spring Boot Starters?

Starters are a set of convenient dependency descriptors which we can include in our application.

Spring Boot provides built-in starters which makes development easier and rapid. For example, if we want to get started using Spring and JPA for database access, just include the **spring-boot-starter-data-jpa** dependency in your project.

For more information click here.

12) What is Spring Boot Actuator?

Spring Boot provides actuator to monitor and manage our application. Actuator is a tool which has HTTP endpoints. when application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.

For more information click here.

13) What is thymeleaf?

It is a server side Java template engine for web application. It's main goal is to bring elegant natural templates to your web application.

It can be integrate with Spring Framework and ideal for HTML5 Java web applications.

For more information click here.

14) How to use thymeleaf?

In order to use Thymeleaf we must add it into our pom.xml file like:

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

For more information click here.

15) How to connect Spring Boot to the database using JPA?

Spring Boot provides **spring-boot-starter-data-jpa** starter to connect Spring application with relational database efficiently. You can use it into project POM (Project Object Model) file.

For more information click here.

16) How to connect Spring Boot application to database using JDBC?

Spring Boot provides starter and libraries for connecting to our application with JDBC. Here, we are creating an application which connects with Mysql database. It includes the following steps to create and setup JDBC with Spring Boot.

For more information click here.

17) What is @RestController annotation in Spring Boot?

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class. We need to import org.springframework.web.bind.annotation package in our file, in order to implement it.

For more information click here.

18) What is @RequestMapping annotation in Spring Boot?

The **@RequestMapping** annotation is used to provide routing information. It tells to the Spring that any HTTP request should map to the corresponding method. We need to import org.springframework.web.annotation package in our file.

For more information click here.

19) How to create Spring Boot application using Spring Starter Project Wizard?

There is one more way to create Spring Boot project in STS (Spring Tool Suite). Creating project by using IDE is always a convenient way. Follow the following steps in order to create a Spring Boot Application by using this wizard.

For more information click here.

20) Spring Vs Spring Boot?

Spring is a web application framework based on Java. It provides tools and libraries to create a complete cutomized web application.

Wheras Spring Boot is a spring module which is used to create spring application project that can just run.

*Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled. To understand the DI better, Let's understand the Dependency Lookup (DL) first:

Dependency Lookup

The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

```
A obj = new AImpl();
```

In such way, we get the resource(instance of A class) directly by new keyword. Another way is factory method:

```
A obj = A.getA();
```

This way, we get the resource (instance of A class) by calling the static factory method getA().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) as:

```
Context ctx = new InitialContext();
Context environmentCtx = (Context) ctx.lookup("java:comp/env");
A obj = (A)environmentCtx.lookup("A");
```

There can be various ways to get the resource to obtain the resource. Let's see the problem in this approach.

Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

- **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
- **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

```
class Employee{
Address address;

Employee(Address address){
this.address=address;
}
public void setAddress(Address address){
this.address=address;
}

}
```

In such case, instance of Address class is provided by external souce such as XML file either by constructor or setter method.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

- By Constructor
- By Setter method

*JavaBean

A JavaBean is a Java class that should follow the following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Simple example of JavaBean class

```java
//Employee.java

package mypack;
public class Employee implements java.io.Serializable{
private int id;
private String name;
public Employee(){}
public void setId(int id){this.id=id;}
public int getId(){return id;}
public void setName(String name){this.name=name;}
public String getName(){return name;}
}
```

How to access the JavaBean class?

To access the JavaBean class, we should use getter and setter methods.

```
package mypack;
public class Test{
public static void main(String args[]){
Employee e=new Employee();//object is created
e.setName("Arjun");//setting value to the object
System.out.println(e.getName());
}}
```

Note: There are two ways to provide values to the object. One way is by constructor and second is by setter method.

---

*JavaBean Properties

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.

* Spring Boot Annotations

Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide **supplemental** information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

In this section, we are going to discuss some important **Spring Boot Annotation** that we will use later in this tutorial.

Core Spring Framework Annotations

@Required: It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

Example

```
public class Machine
{
private Integer cost;
@Required
public void setCost(Integer cost)
{
this.cost = cost;
}
public Integer getCost()
{
return cost;
}
}
```

@Autowired: Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

Example

```java
@Component
public class Customer
{
private Person person;
@Autowired
public Customer(Person person)
{
this.person=person;
}
}
```

@Configuration: It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

Example

```java
@Configuration
public class Vehicle
{
@BeanVehicle engine()
{
return new Vehicle();
}
}
```

@ComponentScan: It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

Example

```
@ComponentScan(basePackages = "com.javatpoint")
@Configuration
public class ScanComponent
{
// ...
}
```

@Bean: It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

Example

```
@Bean
public BeanExample beanExample()
{
return new BeanExample ();
}
```

Spring Framework Stereotype Annotations

@Component: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

Example

```
@Component
public class Student
{
.......
}
```

@Controller: The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

Example

```
@Controller
@RequestMapping("books")
public class BooksController
{
@RequestMapping(value = "/{name}", method = RequestMethod.GET)
public Employee getBooksByName()
{
return booksTemplate;
}
}
```

@Service: It is also used at class level. It tells the Spring that class contains the **business logic**.

Example

```
package com.javatpoint;
@Service
```

```
public class TestService
{
public void service1()
{
//business code
}
}
```

@Repository: It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.

```
package com.javatpoint;
@Repository
public class TestRepository
{
public void delete()
{
//persistence code
}
}
```

Spring Boot Annotations

- **@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.

- **@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

Spring MVC and REST Annotations

- **@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.

Example

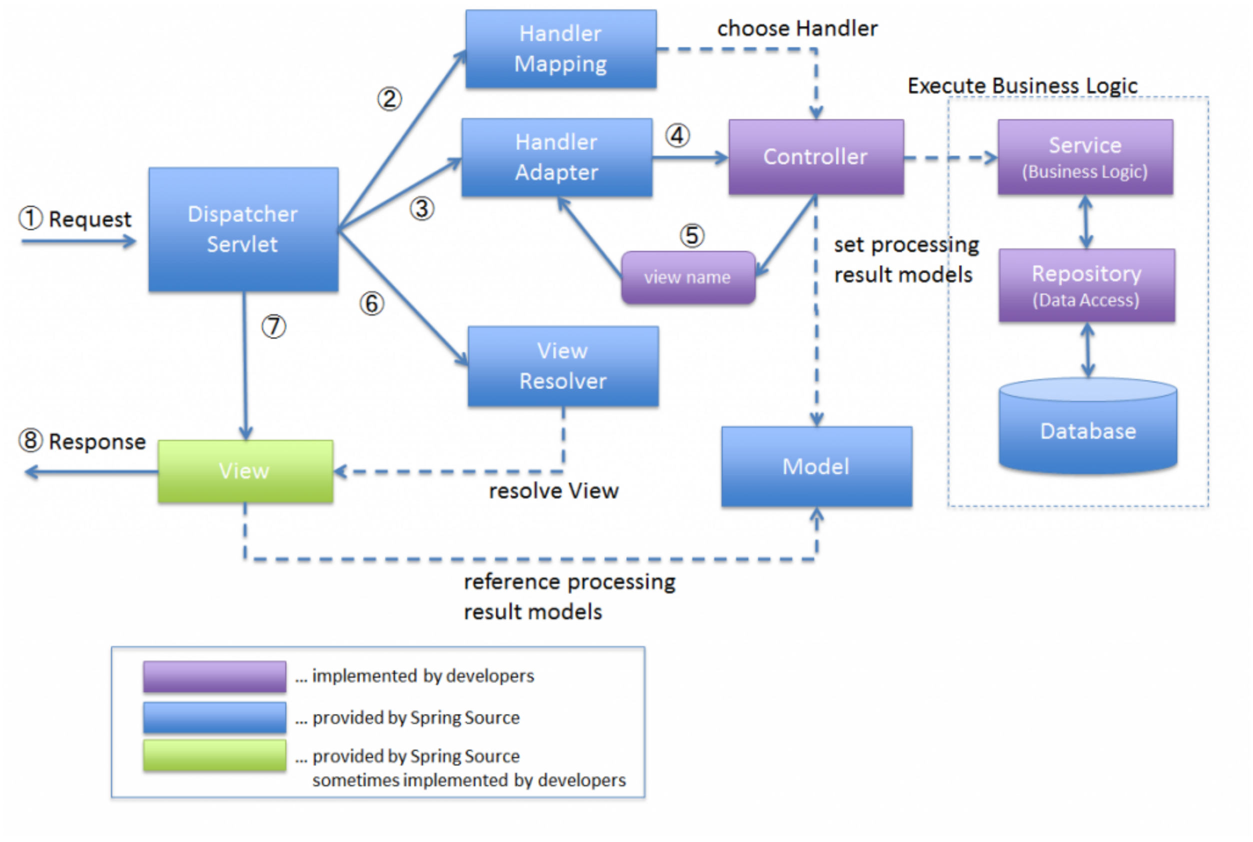
```
@Controller
public class BooksController
{
@RequestMapping("/computer-science/books")
public String getAllBooks(Model model)
{
//application code
return "bookList";
}
```

- **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
- **@PostMapping:** It maps the **HTTP POST** requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
- **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**

- **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes** a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
- **@PatchMapping:** It maps the **HTTP PATCH** requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
- **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
- **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
- **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
- **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
- **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.** For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
- **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody** annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.

- **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

MVC ARCHITECTURE:


Handler Mapping
choose Handler
Execute Business Logic
②
Handler Adapter
④
Controller
Service (Business Logic)
① Request
Dispatcher Servlet
③
⑤
view name
set processing result models
Repository (Data Access)
⑦
⑥
View Resolver
Database
⑧ Response
View
resolve View
Model
reference processing result models
... implemented by developers
... provided by Spring Source
... provided by Spring Source sometimes implemented by developers