**Q=What is Spring Boot?**

**Ans=**First of all Spring Boot is not a framework, it is a way to ease to create stand-alone application with minimal or zero configurations. It is approach to develop spring based application with very less configuration.

**2. What are the advantages of using Spring Boot?**

* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
* It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
* It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
* It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly.
* It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
* It provides lots of plugins to work with embedded and in-memory Databases very easily.

6**. Explain CommandLineRunner interface in Spring boot.**

In spring boot application we can execute any task just before spring boot finishes its startup. To do so we need to create spring bean using CommandLineRunner or ApplicationRunner interface and spring boot will automatically detect them. Both the interfaces have run() method that needs to be overridden in implementing class and make the class as bean by using spring stereotype such as @Component. CommandLineRunner and ApplicationRunner serve the same purpose. The difference between CommandLineRunner and ApplicationRunner is that the run() method of CommandLineRunner accepts array of String as an argument and run() method of ApplicationRunner accepts spring ApplicationArguments as an argument. The arguments which we pass to main() method while starting spring boot, can be accessed in the run() method of CommandLineRunner and ApplicationRunner implementation classes. We can create more than one bean of CommandLineRunner and ApplicationRunner implementing classes. To execute them in an order, we use spring @Order annotation or Ordered interface.

7. **Difference between CommandLineRunner and ApplicationRunner in Spring Boot.**

Both of them provides the same functionality and the only difference between **CommandLineRunner** and **ApplicationRunner** is **CommandLineRunner.run()** accepts **String array[]** whereas **ApplicationRunner.run()** accepts **ApplicationArguments** as argument.

**@Order Annotation Spring Boot**

Whenever you have more than one class implementing the **CommandLineRunner / ApplicationRunner**, then you can use the **@Order** annotation mention which **run()** method has to be executed first.

@RestController

@SpringBootApplication

@Order(1)

**public** **class** Example **implements** ApplicationRunner , CommandLineRunner{

@RequestMapping("/")

String home() {

**return** "Hello World!";

}

@RequestMapping("/name")

String name() {

**return** "my name is rahul";

}

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

System.***out***.println("start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

ConfigurableApplicationContext context = SpringApplication.*run*(Example.**class**, args);

Message msg = context.getBean(Message.**class**);

System.***out***.println(msg.getMessage());

}

**public** **void** run(ApplicationArguments arg0) **throws** Exception {

System.***out***.println("ApplicationRunner &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&& ");

System.***out***.println("Your application started with option names : {}"+ arg0.getOptionNames());

// **TODO** Auto-generated method stub

}

**public** **void** run(String... arg0) **throws** Exception {

System.***out***.println("CommandLineRunner ################################################################################33 ");

System.***out***.println(arg0[0]);

}

}

@Order(0)

@Component

**public** **class** Second **implements** CommandLineRunner{

**public** **void** run(String... args) **throws** Exception {

System.***out***.println("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@");

System.***out***.println(args.length);

}

}

@Service

**public** **class** Message {

**public** String getMessage(){

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

}

}

@Order(2)

@Component

**public** **class** Third **implements** CommandLineRunner{

**public** **void** run(String... args) **throws** Exception {

System.***out***.println("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@");

System.***out***.println(args.length);

}

}

**Q=** **diffrence between @EnableAutoConfiguration and @SpringBootApplication?**

**Ans=**

Many Spring Boot developers always have their main class annotated with @Configuration, @EnableAutoConfiguration and @ComponentScan. Since these annotations are so frequently used together (especially if you follow the [best practices](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-structuring-your-code.html) above), Spring Boot provides a convenient @SpringBootApplicationalternative.

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes:

**package** com.example.myproject;

**import** org.springframework.boot.SpringApplication;

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

*@SpringBootApplication* *// same as @Configuration @EnableAutoConfiguration @ComponentScan*

**public** **class** Application {

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

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

}

}

[**https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html**](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html)

**Q=** **How to scan base package using @SpringBootApplication?**

**Ans=** You can override, with the @SpringBootApplication, the default values of component scan. You just need to include it as a parameters:

@SpringBootApplication(scanBasePackages = "entertainment")

or String array:

@SpringBootApplication(scanBasePackages = {"entertainment", "readinglist"})

# **Q= Spring Boot @ConfigurationProperties ?**

## Ans= 1. Simple Properties file

Normally, you use the @Value annotation to inject the .properties value one by one, this is good for small and simple structure .properties files.

global.properties

email=test@mkyong.com

thread-pool=12

1.1 @Value example.

GlobalProperties.java

@Component

@PropertySource("classpath:global.properties")

public class GlobalProperties {

@Value("${thread-pool}")

private int threadPool;

@Value("${email}")

private String email;

1.2 @ConfigurationProperties example.

GlobalProperties.java

import org.springframework.boot.context.properties.ConfigurationProperties;

@Component

@PropertySource("classpath:global.properties")

@ConfigurationProperties

public class GlobalProperties {

private int threadPool;

private String email;

//getters and setters

}

2. Complex Properties file

2.1 Review a complex structure .properties file below, how you map the values via @Value annotation?

application.properties

#Logging

logging.level.org.springframework.web=ERROR

logging.level.com.mkyong=DEBUG

#Global

email=test@mkyong.com

thread-pool=10

#App

app.menus[0].title=Home

app.menus[0].name=Home

app.menus[0].path=/

app.menus[1].title=Login

app.menus[1].name=Login

app.menus[1].path=/login

app.compiler.timeout=5

app.compiler.output-folder=/temp/

app.error=/error/

@Component

@ConfigurationProperties("app") // prefix app, find app.\* values

public class AppProperties {

private String error;

private List<Menu> menus = new ArrayList<>();

private Compiler compiler = new Compiler();

public static class Menu {

private String name;

private String path;

private String title;

//getters and setters

@Override

public String toString() {

return "Menu{" +

"name='" + name + '\'' +

", path='" + path + '\'' +

", title='" + title + '\'' +

'}';

}

}

public static class Compiler {

private String timeout;

private String outputFolder;

//getters and setters

@Override

public String toString() {

return "Compiler{" +

"timeout='" + timeout + '\'' +

", outputFolder='" + outputFolder + '\'' +

'}';

}

}

//getters and setters

## Property Validation

One of the handy things that this annotation provides the is the **validation of properties using the JSR-303 format**. This allows for all sorts of neat things like checking that a property is not *null*:

|  |
| --- |
| @NotBlank  private String host; |

We can also check the minimum and maximum length of a *String* property:

|  |
| --- |
| @Length(max = 4, min = 1)  private String authMethod; |

Or enforce the minimum and maximum value of an *Integer* property:

|  |
| --- |
| @Min(1025)  @Max(65536)  private int port; |

And finally, we can also make sure that a property matches a certain pattern by defining a regex for the same. This has been done for email, as an example:

|  |
| --- |
| @Pattern(regexp = "^[a-z0-9.\_%+-]+@[a-z0-9.-]+\\.[a-z]{2,6}$")  private String from; |

This helps us reduce a lot of *if – else* conditions in our code and makes it look much cleaner and concise.

**If any of these validations fail then the main application would fail to start with an *IllegalStateException*** till the incorrect property is corrected.

Also, it is important that we declare getters and setters for each of the properties as they’re used by the validator framework to access the concerned properties.

**Placeholder in properties**

**App.name = myApp**

**App.desc= {App.name} is a spring boot application**

**Q=Error Handling in Spring Boot?**

**Ans=**Spring boot provides /error mapping by default that handles all error in sensible way, and it resisters as a global error page in servlet container.

Q=how to change tomcat port number in spring boot?

Ans=change the server.port = 9090 in application.properties.

Q=**how to add debug level of loger in spring boot?**

Ans=add

Logging.level.root=debug in application.properties

Here we can change any logging lavel if we write logging.levele.root= off

Then whole logging will off. But this is not recommended.

->we can set logging level of any specific technology after write Logging.level help will come for ex.

Logging.level.org.springframework.web = info

Etc

We can specify logging pattern also

Logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n

Logging.pattern.file==%d{yyyy-MM-dd HH:mm:ss} – {%thread}%- 5level % logger {36} - %msg%

Logging.path=logs

Q=**What is Spring Boot Actuator?**

Ans= Actuator endpoints allows you to monitor and interact with you application. Spring boot includes number of built in end points and you can also add your own. For example, health endpoint will be mapped to /health.

They’re mainly used to **expose different types of information about the running application** – health, metrics, info, dump, env etc.

### **2.1. How to Enable an Actuator**

To start using the existing actuators in Boot – we’ll just need to add the *spring-boot-actuator* dependency to the pom:

|  |  |
| --- | --- |
| 1  2  3  4 | <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-actuator</artifactId>      <version>1.4.2.RELEASE</version>  </dependency>  **3.Endpoints**: Here are some of the most common endpoints Boot provides out of the box:  • /health – Shows application health information (a simple ‘status’ when accessed over an unauthenticated connection or full message details when authenticated). It is not sensitive by default.  • /info – Displays arbitrary application info. Not sensitive by default.  • /metrics – Shows ‘metrics’ information for the current application. It is also sensitive by default.  • /trace – Displays trace information (by default the last few HTTP requests).  **Customizing Existing Endpoints**  Each endpoint can be customized with properties using the following format: endpoints.[endpoint name].[property to customize]  Three properties are available:  id – by which this endpoint will be accessed over HTTP  enabled – if true then it can be accessed otherwise not  sensitive – if true then need the authorization to show crucial information over HTTP  For example, add the following properties will customize the /beans endpoint:  endpoints.beans.id=springbeans  endpoints.beans.sensitive=false  endpoints.beans.enabled=true  we can change the port number and change the service name ex:  change the /env to /myenv and port number 8080 to 9090  management.port=9090  endpoints.env.id=myenv  endpoints.env.sensitive=false  endpoints.env.enabled=true  ex: /health  {  "status": "UP",  "diskSpace": {  "status": "UP",  "total": 256058060800,  "free": 34416943104,  "threshold": 10485760  }  } **A Custom**HealthIndicator @Component  public class HealthCheck implements HealthIndicator {        @Override      public Health health() {          int errorCode = check(); // perform some specific health check          if (errorCode != 0) {              return Health.down()                .withDetail("Error Code", errorCode).build();          }          return Health.up().build();      }        public int check() {          // Your logic to check health          return 0;      }  } |