As stated by the official [Spring Boot documentation](https://spring.io/projects/spring-boot):

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

There is nothing stating that Spring Boot is only meant for web application development as it offers many capabilities around the Spring Framework itself and which does not relate to web applications such as the core container, Spring AOP, Spring JPA...

The second class-level annotation is @SpringBootApplication. This annotation is known as a meta-annotation, it combines @SpringBootConfiguration, @EnableAutoConfiguration and @ComponentScan.

#### The “main” Method

The final part of our application is the main method. This is 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 MyApplication.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.

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

* @EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-auto-configuration.html)
* @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-structuring-your-code.html))
* @Configuration: allow to register extra beans in the context or import additional configuration classes

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes, as shown in the following example:

**package** com.example.myapplication;

**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);

}

}

|  |
| --- |
| [Note] |
| @SpringBootApplication also provides aliases to customize the attributes of @EnableAutoConfiguration and @ComponentScan |

The apparent purpose of @EnableAutoConfiguration is to enable automatic configuration features of the Spring Boot application, which automatically configures things if certain classes are present in classpath e.g., it can configure Thymeleaf TemplateResolver and ViewResolver if Thymeleaf is present in the classpath.

On the other hand, @SpringBootApplication does three things, it allows you to run the Main class as a JAR with an embedded container. It enables Java configuration, and it also enables Component Scanning.

**What is WebApplicationType in Spring Boot?**

WebApplicationType is an enumeration that represent possible types of web application.

NONE: The application is NOT an web application and should not start an embedded web server.

REACTIVE: The application should run as a reactive web application and should start an embedded reactive web server.

SERVLET: The application should run as a servlet-based web application and should start an embedded servlet web server.

We may call setWebApplicationType(WebApplicationType.NONE) , for example, to choose the type as needed.

Yes, Spring Boot supports the development of both web and non-web applications. We need to remove web dependencies from the classpath and the application context to create a non-web application.

# **Spring Boot Application Internal Working.**

Spring does not generate any code automatically and not using any xml configuration file . so spring uses internally pragmatically configuration done by spring boot developer that are provided by jar.  
we are using just pre-configured jar . and those jar available in:

**META-INF**/**spring.factories**

Enable  
Disable

To Enable preconfigured jars we just need to define dependency in pom.xml file.

*‘<’dependency’>’  
‘<’groupId’>’****org.springframework.boot’****<’/groupId’>’  
‘<’artifactId’>’****spring-boot-starter-data-jpa’****<’/artifactId’>’  
‘<’/dependency’>’*

This dependency will load all the jars related to JPA repository and stored into spring.factories.  
you can go to maven dependencies then click and open spring-boot-autoconfigure jar in the last you will see META-INF folder inside this spring.factories here you will find your jar org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration.

Based on **@Conditional**and **@Configuration** :

***@Configuration****(proxyBeanMethods = false)****@ConditionalOnBean****(DataSource.class)****@ConditionalOnClass****(JpaRepository.class)****@ConditionalOnMissingBean****({ JpaRepositoryFactoryBean.class, JpaRepositoryConfigExtension.class })****@ConditionalOnProperty****(prefix = “spring.data.jpa.repositories”, name = “enabled”, havingValue = “true”,  
matchIfMissing = true)****@Import****(JpaRepositoriesRegistrar.class)****@AutoConfigureAfter****({ HibernateJpaAutoConfiguration.class, TaskExecutionAutoConfiguration.class })  
public class JpaRepositoriesAutoConfiguration {*

*}*

**@ConditionalOnBean**(DataSource.class) :  
— — — — — — — — — — — — — — —  
It will serach for the DataSource bean if it is available then only it will enable JpaRepositoriesAutoConfiguration . So this we need to define DataSource related properties into our property file.

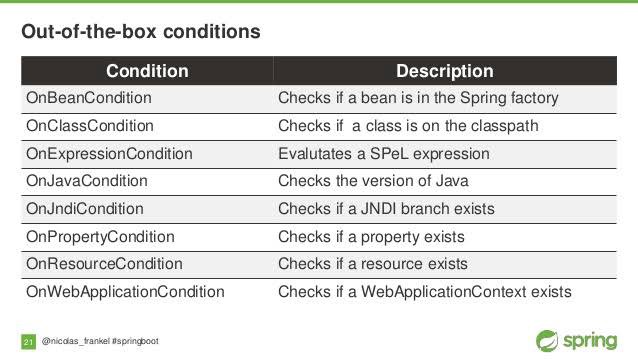
**@ConditionalOnClass**(JpaRepository.class) :  
— — — — — — — — — — — — — — —  
It will serach for the JpaRepository class if it is available then only it will enable JpaRepositoriesAutoConfiguration .

like this :  
**@ConditionalOnMissingBean**({ JpaRepositoryFactoryBean.class, JpaRepositoryConfigExtension.class })  
**@ConditionalOnProperty**(prefix = “spring.data.jpa.repositories”, name = “enabled”, havingValue = “true”, matchIfMissing = true)

If all conditions are true then only it will enable JpaRepositoriesAutoConfiguration class.

**The mainly Conditions checked by spring boot :**

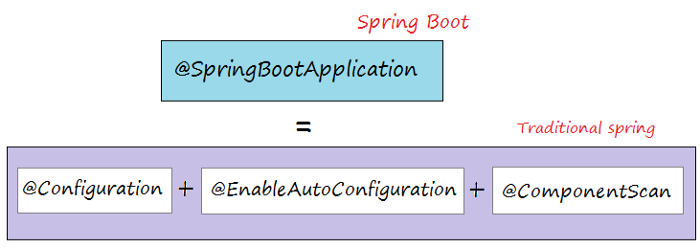




If all the conditions are satisfied then only spring will enable to the component.

**@SpringBootApplication**is the main annotation that we used on our main method and this annotation is the combination of these three annotations :





**High Level Flow Of Spring Boot And How run Method works :**  
================================

From the run method, the main application context is kicked off which in turn searches for the classes annotated with @Configuration, initializes all the declared beans in those configuration classes, and based upon the scope of those beans, stores those beans in JVM, specifically in a space inside JVM which is known as IOC container. After the creation of all the beans, automatically configures the dispatcher servlet and registers the default handler mappings, messageConverts, and all other basic things.

Basically, spring boot supports three embedded servers:- Tomcat (default), Jetty and Undertow.

**run() internal flow :**  
==========

**1.** create application context  
**2.** check Application Type  
**3.** Register the annotated class beans with the context  
**4.** Creates an instance of TomcatEmbeddedServletContainer : and adds the context. Used to deploy our jar automatically.

**open SpringApplication.class :**  
===============

And find here run(String… args) method inside this method you will see the method createApplicationContext() so first it will create application context and inside createApplicationContext() method it will check application type it is SERVLET type Or REACTIVE or DEFAULT context type based on this it will return context.

Now in DEFAULT\_CONTEXT\_CLASS you will see the class **AnnotationConfigApplicationContext.class**

*public****AnnotationConfigApplicationContext****(Class… annotatedClasses) {  
this();  
register(annotatedClasses);  
refresh();  
}*

open this class its constructor is used to Register the annotated class beans with the context.

The classes which are annotated with **@Component, @Service,** **@Configuration** etc. will be register to the context.

And in the finally run(-) method auto deploy the**jar/war** to server.

**@Configuration :**  
It will behave act as bean.  
[**@EnableAutoConfiguart**](http://twitter.com/EnableAutoConfiguart)**ion :**  
it will enable bean based on some condition that we have discussed above.  
[**@ComponentScan**](http://twitter.com/ComponentScan)**:**  
It is mainly used to scan the classes and packages to create the bean.

It is the main class that we need to define to make our spring boot application.

[***@SpringBootApplicatio***](http://twitter.com/SpringBootApplicatio)***n*** *public class Application****{*** *public static void main(String[] args) {  
SpringApplication.run(Application.class, args);  
}****}***

If we will open **@SprinBootApplication** Annotation here you will see it contains :

[***@SpringBootConfigurat***](http://twitter.com/SpringBootConfigurat)***ion***[***@EnableAutoConfigurat***](http://twitter.com/EnableAutoConfigurat)***ion***[***@ComponentScan***](http://twitter.com/ComponentScan)*(excludeFilters = {*[***@Filter***](http://twitter.com/Filter)*(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),*[***@Filter***](http://twitter.com/Filter)*(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })  
public*[*@interface*](http://twitter.com/interface)*SpringBootApplication {  
// code here…..  
}*

### [1.5. Customizing the Bootstrap Configuration](https://docs.spring.io/spring-cloud-commons/docs/current/reference/html/#customizing-the-bootstrap-configuration)

The bootstrap context can be set to do anything you like by adding entries to /META-INF/spring.factories under a key named org.springframework.cloud.bootstrap.BootstrapConfiguration. This holds a comma-separated list of Spring @Configuration classes that are used to create the context. Any beans that you want to be available to the main application context for autowiring can be created here. There is a special contract for @Beans of type ApplicationContextInitializer. If you want to control the startup sequence, you can mark classes with the @Order annotation (the default order is last).

|  |  |
| --- | --- |
|  | When adding custom BootstrapConfiguration, be careful that the classes you add are not @ComponentScanned by mistake into your “main” application context, where they might not be needed. Use a separate package name for boot configuration classes and make sure that name is not already covered by your @ComponentScan or @SpringBootApplication annotated configuration classes. |

The bootstrap process ends by injecting initializers into the main SpringApplication instance (which is the normal Spring Boot startup sequence, whether it runs as a standalone application or is deployed in an application server). First, a bootstrap context is created from the classes found in spring.factories. Then, all @Beans of type ApplicationContextInitializer are added to the main SpringApplication before it is started.

Difference between CommandLineRunner and ApplicationRunner in Spring Boot

These runners are used to run the logic on application startup, for example spring boot has [ApplicationRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/ApplicationRunner.html)(Functional Interface) with run method

ApplicationRunner run() will get execute, just after applicationcontext is created and before spring boot application startup.

ApplicationRunner takes ApplicationArgument which has convenient methods like getOptionNames(), getOptionValues() and getSourceArgs().

And [CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html) is also a Functional interface with run method

CommandLineRunner run() will get execute, just after applicationcontext is created and before spring boot application starts up.

It accepts the argument, which are passed at time of server startup.

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. you can find more information with example [here](https://www.baeldung.com/running-setup-logic-on-startup-in-spring)

CommandLineRunner

This interface provides access to application arguments as string array. Let's see the example code for more clarity.

@Component

public class CommandLineAppStartupRunner implements CommandLineRunner {

private static final Logger logger = LoggerFactory.getLogger(CommandLineAppStartupRunner.class);

@Override

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

logger.info("Application started with command-line arguments: {} . \n To kill this application, press Ctrl + C.", Arrays.toString(args));

}}

ApplicationRunner

ApplicationRunner wraps the raw application arguments and exposes the ApplicationArguments interface, which has many convinent methods to get arguments, like getOptionNames() to return all the arguments' names, getOptionValues() to return the agrument value, and raw source arguments with method getSourceArgs(). Let's see an example:

@Component

public class AppStartupRunner implements ApplicationRunner {

private static final Logger logger = LoggerFactory.getLogger(AppStartupRunner.class);

@Override

public void run(ApplicationArguments args) throws Exception {

logger.info("Your application started with option names : {}", args.getOptionNames());

}}

When to Use It

When you want to execute some piece of code exactly before the application startup completes, you can use it then. In one of our projects, we used these to source data from other microservices via service discovery, which was registered in Consul.

**Using the ApplicationRunner or CommandLineRunner**

If you need to run some specific code once the SpringApplication has started, you can implement the ApplicationRunner or CommandLineRunner interfaces. Both interfaces work in the same way and offer a single run method, which is called just before SpringApplication.run(…​) completes.

This contract is well suited for tasks that should run after application startup but before it starts accepting traffic.

The CommandLineRunner interfaces provides access to application arguments as a string array, whereas the ApplicationRunner uses the ApplicationArguments interface discussed earlier. The following example shows a CommandLineRunner with a run method:

@Component

public class MyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String... args) {

// Do something...

}

}

If several CommandLineRunner or ApplicationRunner beans are defined that must be called in a specific order, you can additionally implement the org.springframework.core.Ordered interface or use the org.springframework.core.annotation.Order annotation.

Application Exit

Each SpringApplication registers a shutdown hook with the JVM to ensure that the ApplicationContext closes gracefully on exit. All the standard Spring lifecycle callbacks (such as the DisposableBean interface or the @PreDestroy annotation) can be used.

In addition, beans may implement the org.springframework.boot.ExitCodeGenerator interface if they wish to return a specific exit code when SpringApplication.exit() is called. This exit code can then be passed to System.exit() to return it as a status code, as shown in the following example:

@SpringBootApplication

public class MyApplication {

@Bean

public ExitCodeGenerator exitCodeGenerator() {

return () -> 42;

}

public static void main(String[] args) {

System.exit(SpringApplication.exit(SpringApplication.run(MyApplication.class, args)));

}}

Also, the ExitCodeGenerator interface may be implemented by exceptions. When such an exception is encountered, Spring Boot returns the exit code provided by the implemented getExitCode() method.

By default, all web endpoints are available beneath the path /actuator with URLs of the form /actuator/{id}. The /actuator base path can be configured by using the management.endpoints.web.base-path property, as shown in the following example:

management.endpoints.web.base-path=/manage

The preceding application.properties example changes the form of the endpoint URLs from /actuator/{id} to /manage/{id}. For example, the URL info endpoint would become /manage/info.

**What are the actuator-provided endpoints used for monitoring the Spring boot application?**

Actuators provide below pre-defined endpoints to monitor our application -

* Health
* Info
* Beans
* Mappings
* Configprops
* Httptrace
* Heapdump
* Threaddump
* Shutdown

**Autoconfiguration**

**9.1. Understanding Auto-configured Beans**

Under the hood, auto-configuration is implemented with standard @Configuration classes. Additional @Conditional annotations are used to constrain when the auto-configuration should apply. Usually, auto-configuration classes use @ConditionalOnClass and @ConditionalOnMissingBean annotations. This ensures that auto-configuration applies only when relevant classes are found and when you have not declared your own @Configuration.

You can browse the source code of [spring-boot-autoconfigure](https://github.com/spring-projects/spring-boot/tree/v2.6.1/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure) to see the @Configuration classes that Spring provides (see the [META-INF/spring.factories](https://github.com/spring-projects/spring-boot/tree/v2.6.1/spring-boot-project/spring-boot-autoconfigure/src/main/resources/META-INF/spring.factories) file).

**9.2. Locating Auto-configuration Candidates**

Spring Boot checks for the presence of a META-INF/spring.factories file within your published jar. The file should list your configuration classes under the EnableAutoConfiguration key, as shown in the following example:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\

com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration

|  |  |
| --- | --- |
|  | Auto-configurations must be loaded that way *only*. Make sure that they are defined in a specific package space and that they are never the target of component scanning. Furthermore, auto-configuration classes should not enable component scanning to find additional components. Specific @Imports should be used instead.  This is a standard component of the Spring Core and it uses SpringFactoriesLoader, get the list of Java Configurations classes configured using the org.springframework.boot.autoconfigure.EnableAutoConfiguration property key. |

You can use the [@AutoConfigureAfter](https://github.com/spring-projects/spring-boot/tree/v2.6.1/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/AutoConfigureAfter.java) or [@AutoConfigureBefore](https://github.com/spring-projects/spring-boot/tree/v2.6.1/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/AutoConfigureBefore.java) annotations if your configuration needs to be applied in a specific order. For example, if you provide web-specific configuration, your class may need to be applied after WebMvcAutoConfiguration.

If you want to order certain auto-configurations that should not have any direct knowledge of each other, you can also use @AutoConfigureOrder. That annotation has the same semantic as the regular @Order annotation but provides a dedicated order for auto-configuration classes.

As with standard @Configuration classes, the order in which auto-configuration classes are applied only affects the order in which their beans are defined. The order in which those beans are subsequently created is unaffected and is determined by each bean’s dependencies and any @DependsOn relationships.

### 9.3. Condition Annotations

You almost always want to include one or more @Conditional annotations on your auto-configuration class. The @ConditionalOnMissingBean annotation is one common example that is used to allow developers to override auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of @Conditional annotations that you can reuse in your own code by annotating @Configuration classes or individual @Bean methods. These annotations include:

* [Class Conditions](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations.class-conditions)
* [Bean Conditions](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations.bean-conditions)
* [Property Conditions](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations.property-conditions)
* [Resource Conditions](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations.resource-conditions)
* [Web Application Conditions](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations.web-application-conditions)
* [SpEL Expression Conditions](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations.spel-conditions)

#### 9.3.1. Class Conditions

The @ConditionalOnClass and @ConditionalOnMissingClass annotations let @Configuration classes be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed by using [ASM](https://asm.ow2.io/), you can use the value attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the name attribute if you prefer to specify the class name by using a String value.

This mechanism does not apply the same way to @Bean methods where typically the return type is the target of the condition: before the condition on the method applies, the JVM will have loaded the class and potentially processed method references which will fail if the class is not present.

To handle this scenario, a separate @Configuration class can be used to isolate the condition, as shown in the following example:

@Configuration(proxyBeanMethods = false)

// Some conditions ...

public class MyAutoConfiguration {

// Auto-configured beans ...

@Configuration(proxyBeanMethods = false)

@ConditionalOnClass(SomeService.class)

public static class SomeServiceConfiguration {

@Bean

@ConditionalOnMissingBean

public SomeService someService() {

return new SomeService();

}

}

}

|  |  |
| --- | --- |
|  | If you use @ConditionalOnClass or @ConditionalOnMissingClass as a part of a meta-annotation to compose your own composed annotations, you must use name as referring to the class in such a case is not handled. |

#### 9.3.2. Bean Conditions

The @ConditionalOnBean and @ConditionalOnMissingBean annotations let a bean be included based on the presence or absence of specific beans. You can use the value attribute to specify beans by type or name to specify beans by name. The search attribute lets you limit the ApplicationContext hierarchy that should be considered when searching for beans.

When placed on a @Bean method, the target type defaults to the return type of the method, as shown in the following example:

@Configuration(proxyBeanMethods = false)

public class MyAutoConfiguration {

@Bean

@ConditionalOnMissingBean

public SomeService someService() {

return new SomeService();

}

}

In the preceding example, the myService bean is going to be created if no bean of type MyService is already contained in the ApplicationContext.

|  |  |
| --- | --- |
|  | You need to be very careful about the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend using only @ConditionalOnBean and @ConditionalOnMissingBean annotations on auto-configuration classes (since these are guaranteed to load after any user-defined bean definitions have been added). |
|  | @ConditionalOnBean and @ConditionalOnMissingBean do not prevent @Configuration classes from being created. The only difference between using these conditions at the class level and marking each contained @Bean method with the annotation is that the former prevents registration of the @Configuration class as a bean if the condition does not match. |

|  |  |
| --- | --- |
|  | When declaring a @Bean method, provide as much type information as possible in the method’s return type. For example, if your bean’s concrete class implements an interface the bean method’s return type should be the concrete class and not the interface. Providing as much type information as possible in @Bean methods is particularly important when using bean conditions as their evaluation can only rely upon to type information that is available in the method signature. |

#### 9.3.3. Property Conditions

The @ConditionalOnProperty annotation lets configuration be included based on a Spring Environment property. Use the prefix and name attributes to specify the property that should be checked. By default, any property that exists and is not equal to false is matched. You can also create more advanced checks by using the havingValue and matchIfMissing attributes.

#### 9.3.4. Resource Conditions

The @ConditionalOnResource annotation lets configuration be included only when a specific resource is present. Resources can be specified by using the usual Spring conventions, as shown in the following example: file:/home/user/test.dat.

#### 9.3.5. Web Application Conditions

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations let configuration be included depending on whether the application is a “web application”. A servlet-based web application is any application that uses a Spring WebApplicationContext, defines a session scope, or has a ConfigurableWebEnvironment. A reactive web application is any application that uses a ReactiveWebApplicationContext, or has a ConfigurableReactiveWebEnvironment.

The @ConditionalOnWarDeployment annotation lets configuration be included depending on whether the application is a traditional WAR application that is deployed to a container. This condition will not match for applications that are run with an embedded server.

@ConditionalOnWebApplication and @ConditionalOnNotWebApplication

With these annotations, we can create conditions based on if the current application is or isn't a web application:

@ConditionalOnWebApplication

HealthCheckController healthCheckController() {

#### 9.3.6. SpEL Expression Conditions

The @ConditionalOnExpression annotation lets configuration be included based on the result of a [SpEL expression](https://docs.spring.io/spring-framework/docs/5.3.13/reference/html/core.html#expressions).

There seems to be a change in the way configurations beans are detected by Spring since version 2.7, it seems to be covered in this migration [guide](https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#auto-configuration-files)

Basically, instead of providing configuration in META-INF/spring.factories, you would do so in META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

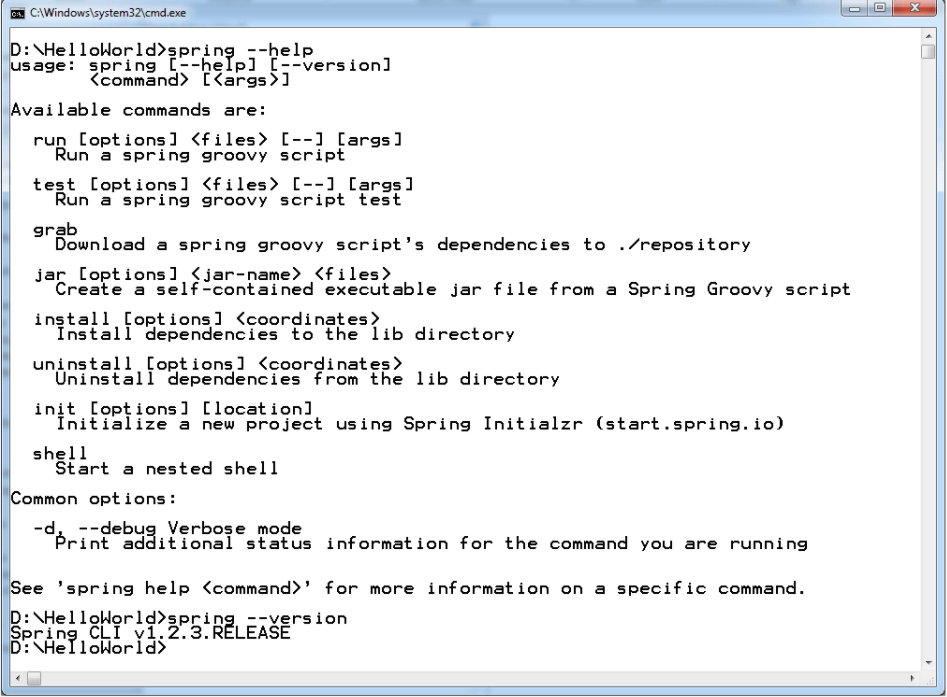
**Sample of spring.factories**

*# Logging Systems*org.springframework.boot.logging.LoggingSystemFactory=\  
org.springframework.boot.logging.logback.LogbackLoggingSystem.Factory,\  
org.springframework.boot.logging.log4j2.Log4J2LoggingSystem.Factory,\  
org.springframework.boot.logging.java.JavaLoggingSystem.Factory  
  
*# PropertySource Loaders*org.springframework.boot.env.PropertySourceLoader=\  
org.springframework.boot.env.PropertiesPropertySourceLoader,\  
org.springframework.boot.env.YamlPropertySourceLoader  
  
*# ConfigData Location Resolvers*org.springframework.boot.context.config.ConfigDataLocationResolver=\  
org.springframework.boot.context.config.ConfigTreeConfigDataLocationResolver,\  
org.springframework.boot.context.config.StandardConfigDataLocationResolver  
  
*# ConfigData Loaders*org.springframework.boot.context.config.ConfigDataLoader=\  
org.springframework.boot.context.config.ConfigTreeConfigDataLoader,\  
org.springframework.boot.context.config.StandardConfigDataLoader  
  
*# Application Context Factories*org.springframework.boot.ApplicationContextFactory=\  
org.springframework.boot.web.reactive.context.ReactiveWebServerApplicationContextFactory,\  
org.springframework.boot.web.servlet.context.ServletWebServerApplicationContextFactory  
  
*# Run Listeners*org.springframework.boot.SpringApplicationRunListener=\  
org.springframework.boot.context.event.EventPublishingRunListener  
  
*# Error Reporters*org.springframework.boot.SpringBootExceptionReporter=\  
org.springframework.boot.diagnostics.FailureAnalyzers  
  
*# Application Context Initializers*org.springframework.context.ApplicationContextInitializer=\  
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\  
org.springframework.boot.context.ContextIdApplicationContextInitializer,\  
org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\  
org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitializer,\  
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer  
  
*# Application Listeners*org.springframework.context.ApplicationListener=\  
org.springframework.boot.ClearCachesApplicationListener,\  
org.springframework.boot.builder.ParentContextCloserApplicationListener,\  
org.springframework.boot.context.FileEncodingApplicationListener,\  
org.springframework.boot.context.config.AnsiOutputApplicationListener,\  
org.springframework.boot.context.config.DelegatingApplicationListener,\  
org.springframework.boot.context.logging.LoggingApplicationListener,\  
org.springframework.boot.env.EnvironmentPostProcessorApplicationListener  
  
*# Environment Post Processors*org.springframework.boot.env.EnvironmentPostProcessor=\  
org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor,\  
org.springframework.boot.context.config.ConfigDataEnvironmentPostProcessor,\  
org.springframework.boot.env.RandomValuePropertySourceEnvironmentPostProcessor,\  
org.springframework.boot.env.SpringApplicationJsonEnvironmentPostProcessor,\  
org.springframework.boot.env.SystemEnvironmentPropertySourceEnvironmentPostProcessor,\  
org.springframework.boot.reactor.DebugAgentEnvironmentPostProcessor  
  
*# Failure Analyzers*org.springframework.boot.diagnostics.FailureAnalyzer=\  
org.springframework.boot.context.config.ConfigDataNotFoundFailureAnalyzer,\  
org.springframework.boot.context.properties.IncompatibleConfigurationFailureAnalyzer,\  
org.springframework.boot.context.properties.NotConstructorBoundInjectionFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.BeanCurrentlyInCreationFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.BeanDefinitionOverrideFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.BeanNotOfRequiredTypeFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.BindFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.BindValidationFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.UnboundConfigurationPropertyFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.MutuallyExclusiveConfigurationPropertiesFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.NoSuchMethodFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.NoUniqueBeanDefinitionFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.PortInUseFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.ValidationExceptionFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.InvalidConfigurationPropertyNameFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.InvalidConfigurationPropertyValueFailureAnalyzer,\  
org.springframework.boot.diagnostics.analyzer.PatternParseFailureAnalyzer,\  
org.springframework.boot.liquibase.LiquibaseChangelogMissingFailureAnalyzer,\  
org.springframework.boot.web.context.MissingWebServerFactoryBeanFailureAnalyzer,\  
org.springframework.boot.web.embedded.tomcat.ConnectorStartFailureAnalyzer  
  
*# Failure Analysis Reporters*org.springframework.boot.diagnostics.FailureAnalysisReporter=\  
org.springframework.boot.diagnostics.LoggingFailureAnalysisReporter  
  
*# Database Initializer Detectors*org.springframework.boot.sql.init.dependency.DatabaseInitializerDetector=\  
org.springframework.boot.flyway.FlywayDatabaseInitializerDetector,\  
org.springframework.boot.jdbc.init.DataSourceScriptDatabaseInitializerDetector,\  
org.springframework.boot.liquibase.LiquibaseDatabaseInitializerDetector,\  
org.springframework.boot.orm.jpa.JpaDatabaseInitializerDetector,\  
org.springframework.boot.r2dbc.init.R2dbcScriptDatabaseInitializerDetector  
  
*# Depends On Database Initialization Detectors*org.springframework.boot.sql.init.dependency.DependsOnDatabaseInitializationDetector=\  
org.springframework.boot.sql.init.dependency.AnnotationDependsOnDatabaseInitializationDetector,\  
org.springframework.boot.jdbc.SpringJdbcDependsOnDatabaseInitializationDetector,\  
org.springframework.boot.jooq.JooqDependsOnDatabaseInitializationDetector,\  
org.springframework.boot.orm.jpa.JpaDependsOnDatabaseInitializationDetector

### What are the most common Spring Boot CLI commands?

-run, -test, -grap, -jar, -war, -install, -uninstall, --init, -shell, -help.

To check the description, run spring --help from the terminal.

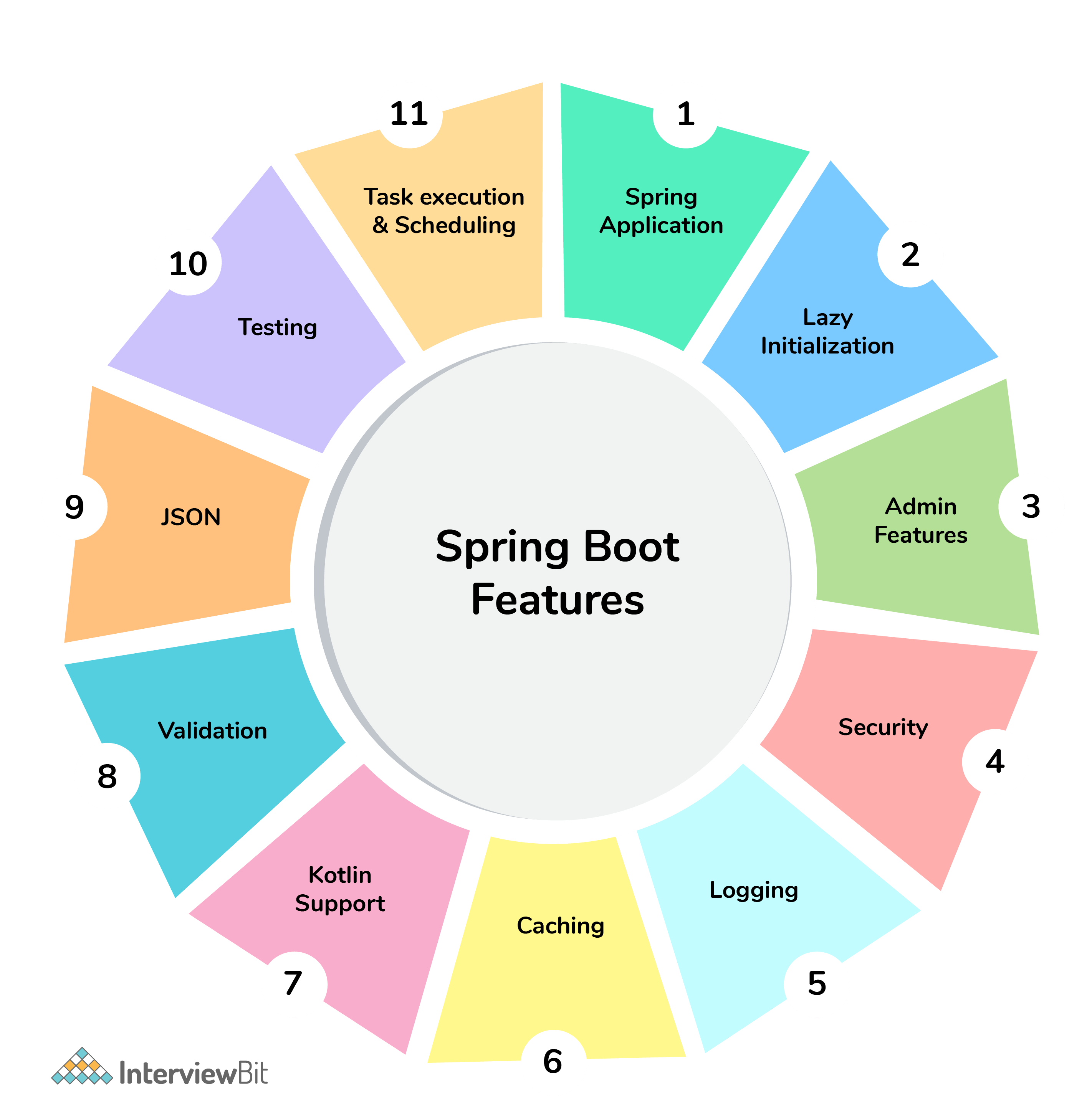
Spring Boot CLI Commands

## What is Spring boot?

Sprint boot is a Java-based spring framework used for Rapid Application Development (to build stand-alone microservices). It has extra support of auto-configuration and embedded application server like tomcat, jetty, etc.

## Features of Spring Boot that make it different?

* Creates stand-alone spring application with minimal configuration needed.
* It has embedded tomcat, jetty which makes it just code and run the application.
* Provide production-ready features such as metrics, health checks, and externalized configuration.
* Absolutely no requirement for XML configuration.

Spring Boot Features

## Spring Boot Interview Questions For Freshers

### 1. What are the advantages of using Spring Boot?

The advantages of Spring Boot are listed below:

* Easy to understand and develop spring applications.
* Spring Boot is nothing but an existing framework with the addition of an embedded HTTP server and annotation configuration which makes it easier to understand and faster the process of development.
* Increases productivity and reduces development time.
* Minimum configuration.
* We don’t need to write any XML configuration, only a few annotations are required to do the configuration.

### 2. What are the Spring Boot key components?

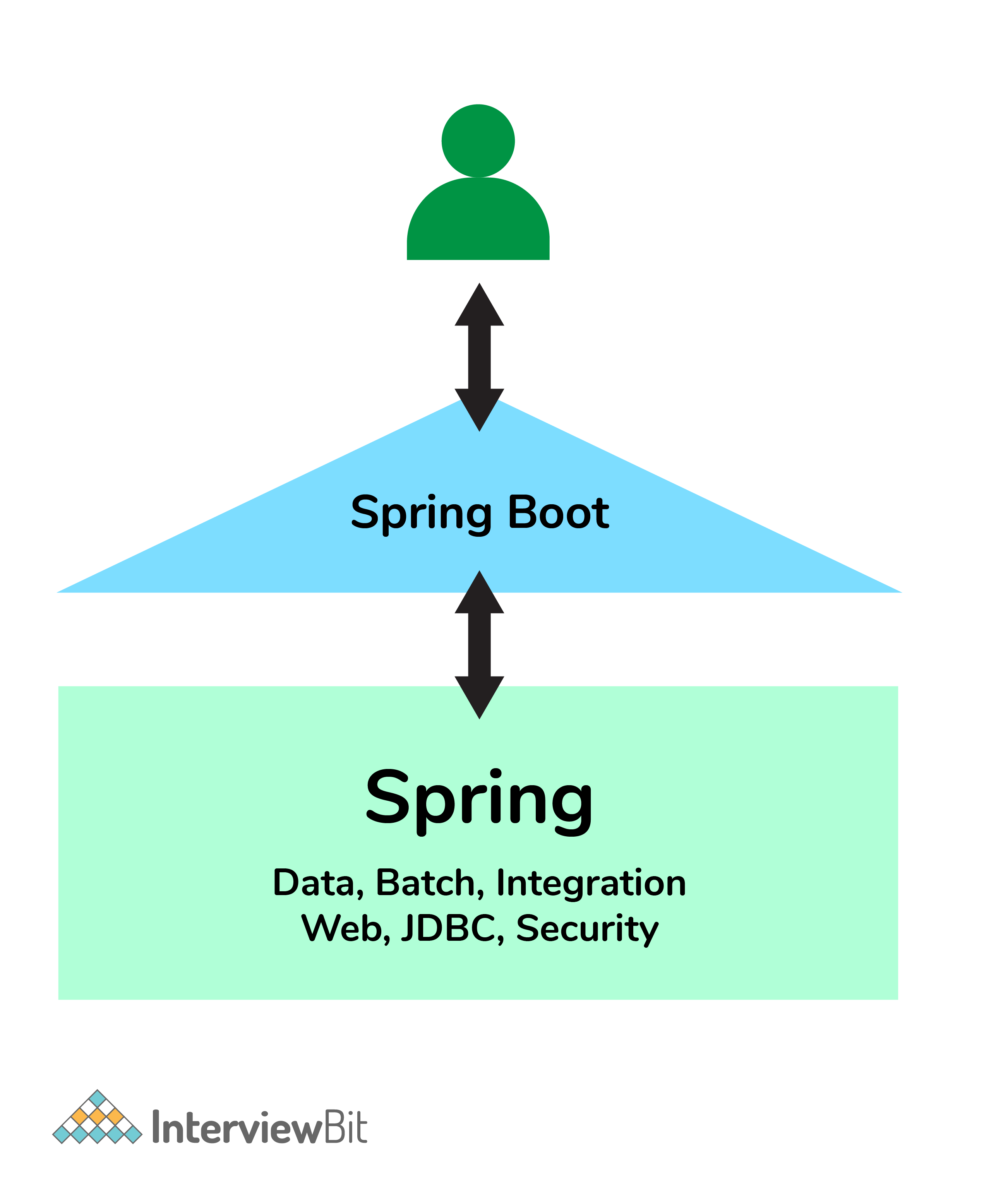
Below are the four key components of spring-boot:

* Spring Boot auto-configuration.
* Spring Boot CLI.
* Spring Boot starter POMs.
* Spring Boot Actuators.

### 3. Why Spring Boot over Spring?

Below are some key points which spring boot offers but spring doesn’t:

* Starter POM.
* Version Management.
* Auto Configuration.
* Component Scanning.
* Embedded server.
* InMemory DB.
* Actuators

Spring Boot simplifies the spring feature for the user:

### 4. What is the starter dependency of the Spring boot module?

Spring boot provides numbers of starter dependency, here are the most commonly used -

* Data JPA starter.
* Test Starter.
* Security starter.
* Web starter.
* Mail starter.
* Thymeleaf starter.

### 5. How does Spring Boot works?

Spring Boot automatically configures your application based on the dependencies you have added to the project by using annotation. The entry point of the spring boot application is the class that contains @SpringBootApplication annotation and the main method.

Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.

### 6. What does the @SpringBootApplication annotation do internally?

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes. Spring Boot enables the developer to use a single annotation instead of using multiple. But, as we know, Spring provided loosely coupled features that we can use for each annotation as per our project needs.

### 7. What is the purpose of using @ComponentScan in the class files?

Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the @ComponentScan annotation for your class file to scan your components added to your project.

### 8. How does a spring boot application get started?

Just like any other Java program, a Spring Boot application must have a main method. This method serves as an entry point, which invokes the SpringApplication#run method to bootstrap the application.

@SpringBootApplication

**public** **class** **MyApplication** {

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

SpringApplication.run(MyApplication.class);

// other statements

}

}

### 9.

## **Custom Starter with Spring Boot**

To create our own custom starter, we require following components

* The auto-configure module with auto configuration class.
* The stater module which will bring all required dependencies using **pom.xml**

For this post, we are creating only a single module combining both auto-configuration code and starter module for getting all required dependencies. We will create a simple hello service stater with following features

1. the hello-service-spring-boot-starter with HelloService which takes the name as input to say hello.
2. HelloService will use the default configuration for the default name.
3. We will create Spring Boot demo application for using our hello-service-starter.

#### 2.1 The Auto-Configure Module

The hello-service-spring-boot-starter will have the following classes and configurations

* HelloServiveProperties file for default name.
* HelloService interface and HelloServiceImpl class.
* HelloServiceAutoConfiguration to create HelloService Bean.
* The ***pom.xml*** file for bringing required dependencies to our custom starter.

#### 2.2 Property and Service Class

package com.javadevjournal.service;

public interface HelloService {

void hello();

}

*//Impl Service*

public class HelloServiceImpl implements HelloService {

@Override

public void hello() {

System.out.println("Hello from the default starter");

}

}

#### 2.3 The AutoConfigure Module and Class

@Configuration

@ConditionalOnClass(HelloService.class)

public class HelloServiceAutoConfiguration {

*//conditional bean creation*

@Bean

@ConditionalOnMissingBean

public HelloService helloService(){

return new HelloServiceImpl();

}

}

The final piece of our auto-configuration is the addition of this class in the spring.factories property file located in the /src/main/resources/META-INF.

org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.javadevjournal.config.HelloServiceAutoConfiguration

On application startup

* HelloServiceAutoConfiguration will run if HelloService class is available in the classpath. ( @ConditionOnClass annotation).
* HelloService Bean will be created by **Spring Boot** if it is not available (@ConditionalOnMissingBean).
* If developer defines their own HelloService bean, our customer starter will not create HelloService Bean.

#### 2.4 The pom.xml

The last part of the custom starter is the pom.xml to bring in all the required dependencies.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"> <modelVersion>4.0.0</modelVersion>

<parent><groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starters</artifactId>

<version>1.5.9.RELEASE</version>

</parent>

<groupId>com.javadevjournal</groupId>

<artifactId>hello-service-spring-boot-starter</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>hello-service-spring-boot-starter</name>

<description>Custom Starter for Spring Boot</description>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-autoconfigure</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId> </plugin> </plugins> </build></project>

Let’s cover some interesting points in the pom.xml

* We defined the parent as spring-boot-starters. We needed to pull in required dependencies.

[pullquote align=”normal”]For more information on parent pom. Please read our article [Spring Boot Starter Parent](https://www.javadevjournal.com/spring-boot/spring-boot-starter-parent/)  [/pullquote]

#### 2.4  Naming Convention

While creating a custom starter with Spring Boot, read below guidelines for the naming convention.

* Your custom starter module should not start with Spring Boot.
* Use name-spring-boot-starter as a guideline. In our case, we named our starter as hello-service-spring-boot-starter.

## **3. Using the custom starter**

Let’s create a [sample Spring Boot application](https://www.javadevjournal.com/spring/introduction-to-spring-boot/) to use our custom starter. Once We create starter app, add the custom starter as a dependency in **pom.xml**.

<dependency>

<groupId>com.javadevjournal</groupId>

<artifactId>hello-service-spring-boot-starter</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

Here is our Spring Boot starter class

@SpringBootApplication

public class CustomStarterAppApplication implements CommandLineRunner {

@Autowired

HelloService service;

public static void main(String[] args) {

SpringApplication.run(CustomStarterAppApplication.class, args);

}

@Override

public void run(String... strings) throws Exception {

service.hello();

}

If we run our application, you will see following output in the console

018-01-23 20:27:52.138 INFO 20441 --- [  main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.....

<em><strong>Hello from the default starter</strong></em>

2018-01-23 20:27:52.620 INFO 20441 --- [  main] c.j.CustomStarterAppApplication  : Started CustomStarterAppApplication in ....

We have defined no HelloService is our demo application. When Spring Boot started, auto-configuration did not find any custom bean definition. Our custom starter auto configuration class created default “HelloService” bean. (as visible from the output).To understand Spring Boot auto-configuration logic and functionality, let’s create a custom HelloService bean in our sample application

public class CustomHelloService implements HelloService {

@Override

public void hello() {

System.out.println("We are overriding our custom Hello Service");

}

}

*//bean bean definition*

@SpringBootApplication

public class CustomStarterAppApplication implements CommandLineRunner {

@Autowired

HelloService service;

public static void main(String[] args) {

SpringApplication.run(CustomStarterAppApplication.class, args);

}

@Override

public void run(String... strings) throws Exception {

service.hello();

}

@Bean

public HelloService helloService(){

return new CustomHelloService();

}

}

Here is the output on running this application

2018-01-23 20:36:48.991 INFO 20529 --- [  main] o.s.j.e.a.AnnotationMBeanExporter  : Registering beans for JMX exposure on startup

<em><strong>We are overriding our custom Hello Service</strong></em>

2018-01-23 20:36:49.000 INFO 20529 --- [  main] c.j.CustomStarterAppApplication  : Started CustomStarterAppApplication in 0.701 seconds

When we defined our custom bean, Spring Boot default HelloService is no longer available. This enables developers to completely override default bean definition by creating/ providing their own bean definition.