# **What is Spring Boot? Why should you use it? How it is diff from spring framework ?**

**Spring –**

1. Is a dependency injection framework.
2. It is basically used to manage the life cycle of java classes (beans). It consists of a lot of boilerplate configuration.
3. Uses XML based configuration.
4. It takes time to have a spring application up and running and it’s mainly because of boilerplate code.

**Spring boot-**

1. It is a suite of pre- configured frameworks and technologies which helps to remove boilerplate configuration.
2. Uses annotations.
3. It is used to create a production-ready code.

Spring Boot is essentially a framework for rapid application development built on top of the Spring Framework. With its auto-configuration and embedded application server support, combined with the extensive documentation and community support it enjoys, Spring Boot is one of the most popular technologies in the Java ecosystem as of date.

Here are a few salient features:

* [**Starters**](https://www.baeldung.com/spring-boot-starters) – a set of dependency descriptors to include relevant dependencies at a go
* [**Auto-configuration**](https://www.baeldung.com/spring-boot-annotations#enable-autoconfiguration) – a way to automatically configure an application based on the dependencies present on the classpath
* [**Actuator**](https://www.baeldung.com/spring-boot-actuators) – to get production-ready features such as monitoring
* [**Security**](https://www.baeldung.com/security-spring)
* [**Logging**](https://www.baeldung.com/spring-boot-logging)

# **What are some important features of using Spring Boot?**

Over the years, Spring has been growing more and more complex, and the amount of configuration such application requires can be intimidating. This is where Spring Boot comes in handy – it makes configuring a Spring application a breeze.

Essentially, while Spring is unopinionated, **Spring Boot takes an opinionated view of the platform and libraries, letting us get started quickly.**

Here are two of the most important benefits Spring Boot brings in:

* Auto-configure applications based on the artifacts it finds on the classpath
* Provide non-functional features common to applications in production, such as security or health checks
* 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.

# **What is Spring Initializer? why should you use it?**

Spring Initializr is a convenient way to create a Spring Boot project.

We can go to the [Spring Initializr](https://start.spring.io/) site, choose a dependency management tool (either Maven or Gradle), a language (Java, Kotlin or Groovy), a packaging scheme (Jar or War), version and dependencies, and download the project.

This **creates a skeleton project for us** and saves setup time so that we can concentrate on adding business logic.

Even when we use our IDE's (such as STS or Eclipse with STS plugin) new project wizard to create a Spring Boot project, it uses Spring Initializr under the hood.

Spring Initializr is easiest way to create a Spring Boot project.

We can go to the Spring Initializr site (https://start.spring.io/), choose below thing and download the project dependency management tool (either Maven or Gradle), a language (Java, Kotlin or Groovy), a packaging scheme (Jar or War), version and dependencies, .

# **What is auto-configuration in Spring boot? how does it help? Why Spring Boot is called opinionated? What is starter dependency in Spring Boot? how does it help?**

Starter dependency,Auto-Configuration,Spring Initializer,Spring Actuator,Spring CLI

Each starter plays a role as a one-stop-shop for all the Spring technologies we need. Other required dependencies are then transitively pulled in and managed in a consistent way.

All starters are under the org.springframework.boot group and their names start with spring-boot-starter-. This naming pattern makes it easy to find starters, especially when working with IDEs that support searching dependencies by name.

At the time of this writing, there are more than 50 starters at our disposal. The most commonly used are:

* spring-boot-starter: core starter, including auto-configuration support, logging, and YAML
* spring-boot-starter-aop: starter for aspect-oriented programming with Spring AOP and AspectJ
* spring-boot-starter-data-jpa: starter for using Spring Data JPA with Hibernate
* spring-boot-starter-security: starter for using Spring Security
* spring-boot-starter-test: starter for testing Spring Boot applications
* spring-boot-starter-web: starter for building web, including RESTful, applications using Spring MVC

# **What is the difference between @SpringBootApplication and @EnableAutoConfiguration annotation?**

@SpringBootApplication is a newer version of @EnableAutoConfiguration which was introduced in Spring Boot 1.2. @SpringBootApplication is a combination of three annotations,

* @Configuration - for java based configuration classes.
* @ComponentScan - to enable component scanning, all the packages and subpackages will be auto-scanned which are under the root package on which @SpringBootApplication is applied.
* @EnableAutoConfiguration - to enable auto-configuration of the classes bases on the jars added in classpath.

**@ComponentScan** enables component scanning so that web controller classes and other components that you create will be automatically discovered and registered as beans in spring's application context. You can specify the base packages that will be scanned for auto-discovering and registering of beans.

One of the optional element is,

* basePackages - can be used to state specific packages to scan.

@ComponentScan(basePackages = {"com.example.test"})

@Configuration

public class SpringConfiguration { }

# **What is the difference between the @ComponentScan and @EnableAutoConfiguration annotations in Spring Boot?**

[@EnableAutoConfiguration](http://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/EnableAutoConfiguration.html) annotation tells Spring Boot to "guess" how you will want to configure Spring, based on the jar dependencies that you have added. For example, If HSQLDB is on your classpath, and you have not manually configured any database connection beans, then Spring will auto-configure an in-memory database.

[@ComponentScan](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) tells Spring to look for other components, configurations, and services in the specified package. Spring is able to auto scan, detect and register your beans or components from pre-defined project package. If no package is specified current class package is taken as the root package.

# [**spring-boot without @SpringBootApplication**](https://stackoverflow.com/questions/33290683/spring-boot-without-springbootapplication)

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/current/reference/html/using.html#using.auto-configuration)
* @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.structuring-your-code))
* @SpringBootConfiguration: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring’s standard @Configuration that aids [configuration detection](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing.spring-boot-applications.detecting-configuration) in your integration tests.

@SpringBootApplication // same as @SpringBootConfiguration @EnableAutoConfiguration

// @ComponentScan

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

Probably you want use @Configuration + @ComponentScan. If you want load xml configuration you can use: @ImportResource annotation.

If you want use autoconfiguration, but you can disable a few auto configurations, eg: @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

None of these features are mandatory and you may choose to replace this single annotation by any of the features that it enables. For instance, you may not want to use component scan or configuration properties scan in your application:

@SpringBootConfiguration(proxyBeanMethods = false)

@EnableAutoConfiguration

@Import({ SomeConfiguration.class, AnotherConfiguration.class })

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

In this example, MyApplication is just like any other Spring Boot application except that @Component-annotated classes and @ConfigurationProperties-annotated classes are not detected automatically and the user-defined beans are imported explicitly (see @Import).

# **How to Register a Custom Auto-Configuration?**

To register an auto-configuration class, we must have its fully-qualified name listed under the EnableAutoConfiguration key in the META-INF/spring.factories file:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.baeldung.autoconfigure.CustomAutoConfiguration

If we build a project with Maven, that file should be placed in the resources/META-INF directory, which will end up in the mentioned location during the package phase.

# **How to disable a specific auto-configuration class?**

If we want to disable a specific auto-configuration, we can indicate it using the *exclude* attribute of the *@EnableAutoConfiguration* annotation. For instance, this code snippet neutralizes *DataSourceAutoConfiguration*:

// other annotations

@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)

**public** **class** **MyConfiguration** { }

If we enabled auto-configuration with the *@SpringBootApplication* annotation — which has *@EnableAutoConfiguration* as a meta-annotation — we could disable auto-configuration with an attribute of the same name:

// other annotations

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

**public** **class** **MyConfiguration** { }

We can also disable an auto-configuration with the *spring.autoconfigure.exclude* environment property. This setting in the *application.properties* file does the same thing as before:

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

# **How to Tell an Auto-Configuration to Back Away When a Bean Exists**

To instruct an auto-configuration class to back off when a bean is already existent, we can use the @ConditionalOnMissingBean annotation. The most noticeable attributes of this annotation are:

* value: The types of beans to be checked
* name: The names of beans to be checked

When placed on a method adorned with @Bean, the target type defaults to the method's return type:

@Configuration

**public** **class** **CustomConfiguration** {

@Bean

@ConditionalOnMissingBean

**public** CustomService **service**() { ... }

}

# **How to Deploy Spring Boot Web Applications as Jar and War Files?**

Traditionally, we package a web application as a WAR file, then deploy it into an external server. Doing this allows us to arrange multiple applications on the same server. During the time that CPU and memory were scarce, this was a great way to save resources.

However, things have changed. Computer hardware is fairly cheap now, and the attention has turned to server configuration. A small mistake in configuring the server during deployment may lead to catastrophic consequences.

**Spring tackles this problem by providing a plugin, namely** *spring-boot-maven-plugin***, to package a web application as an executable JAR**. To include this plugin, just add a *plugin* element to *pom.xml*:

<**plugin**>

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

<**artifactId**>spring-boot-maven-plugin</**artifactId**>

</**plugin**>

With this plugin in place, we'll get a fat JAR after executing the *package* phase. This JAR contains all the necessary dependencies, including an embedded server. Thus, we no longer need to worry about configuring an external server.

We can then run the application just like we would an ordinary executable JAR.

Notice that the *packaging* element in the *pom.xml* file must be set to *jar* to build a JAR file:

<**packaging**>jar</**packaging**>

If we don't include this element, it also defaults to jar.

In case we want to build a WAR file, change the packaging element to war:

<**packaging**>war</**packaging**>

And leave the container dependency off the packaged file:

<**dependency**>

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

<**artifactId**>spring-boot-starter-tomcat</**artifactId**>

<**scope**>provided</**scope**>

</**dependency**>

After executing the Maven package phase, we'll have a deployable WAR file.

For war - specifiy the packaging and excude the tomcat embeded server

* make depencency - provided

<dependency>

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

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

* make the @springbootapplication class - extends springbootservletinitializer
* pom.xml - <packaging>war</packaging> and <build><finalName>${artifactId}</finalName> </build> = for generate the war name
* eclipse "clean package or maven install" and copy the war into tomcat/webapps
* start the server and type url = http://localhost:8080/warName for context)

# **How to Use Spring Boot for Command Line Applications?**

Just like any other Java program, a Spring Boot command line 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

}

}

The SpringApplication class then fires up a Spring container and auto-configures beans.

Notice we must pass a configuration class to the run method to work as the primary configuration source. By convention, this argument is the entry class itself.

After calling the run method, we can execute other statements as in a regular program.

# **What Does it Mean that Spring Boot Supports Relaxed Binding**

Relaxed binding in Spring Boot is applicable to [the type-safe binding of configuration properties](https://www.baeldung.com/configuration-properties-in-spring-boot).

With relaxed binding, **the key of a property doesn't need to be an exact match of a property name.** Such an environment property can be written in camelCase, kebab-case, snake\_case, or in uppercase with words separated by underscores.

For example, if a property in a bean class with the @ConfigurationProperties annotation is named myProp, it can be bound to any of these environment properties: myProp, my-prop, my\_prop, or MY\_PROP.

# **What is the use of Profiles in spring boot?**

When developing applications for the enterprise, we typically deal with multiple environments such as Dev, QA, and Prod. The configuration properties for these environments are different.

For example, we might be using an embedded H2 database for Dev, but Prod could have the proprietary Oracle or DB2. Even if the DBMS is the same across environments, the URLs would definitely be different.

To make this easy and clean, **Spring has the provision of profiles, to help separate the configuration for each environment**. So that instead of maintaining this programmatically, the properties can be kept in separate files such as application-dev.*properties* and application-prod.properties. The default application.properties points to the currently active profile using spring.*profiles*.*active* so that the correct configuration is picked up.

application.properties - default (spring.profiles.active=dev spcificy to pick correct properties)

application-uat.properties

application-dev.properties

# **SPRING\_CONFIG\_LOCATION**

Developers can use YAML files, properties files, command-line arguments, and environment variables to externalize configuration.

java -jar bootjpaoracle-0.0.1-SNAPSHOT.jar (use internal properties)

--Command Line - setting external properties for JAR

java -jar bootjpaoracle-0.0.1-SNAPSHOT.jar --server.servlet.context-path=/inventory-uat (Override the specific internal properties)

java -jar bootjpaoracle-0.0.1-SNAPSHOT.jar --spring.config.location=./application.properties (Override all properties)

environment vairable

SPRING\_CONFIG\_LOCATION=file:C:\data\application.properties

java -jar bootjpaoracle-0.0.1-SNAPSHOT.jar

# **What is Spring Actuator? What are its advantages? How to enable Actuator in Spring boot application?**

Essentially, Actuator brings Spring Boot applications to life by enabling production-ready features. **These features allow us to monitor and manage applications when they're running in production**.

Integrating Spring Boot Actuator into a project is very simple. All we need to do is to include the *spring-boot-starter-actuator* starter in the *pom.xml* file:

<**dependency**>

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

<**artifactId**>spring-boot-starter-actuator</**artifactId**>

</**dependency**>

Spring Boot Actuator can expose operational information using either HTTP or JMX endpoints. Most applications go for HTTP, though, where the identity of an endpoint and the */actuator* prefix form a URL path.

Here are some of the most common built-in endpoints Actuator provides:

* *env:* Exposes environment properties
* *health:* Shows application health information
* *httptrace:* Displays HTTP trace information
* *info:* Displays arbitrary application information
* *metrics:* Shows metrics information
* *loggers:* Shows and modifies the configuration of loggers in the application
* *mappings:* Displays a list of all *@RequestMapping* paths
* Health
* Info
* Beans
* Mappings
* Configprops
* Httptrace
* Heapdump
* Threaddump
* Shutdown

# **What Are the Basic Annotations that Spring Boot Offers?**

The primary annotations that Spring Boot offers reside in its org.springframework.boot.autoconfigure and its sub-packages. Here are a couple of basic ones:

* @EnableAutoConfiguration – to make Spring Boot look for auto-configuration beans on its classpath and automatically apply them.
* @SpringBootApplication – used to denote the main class of a Boot Application. This annotation combines @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.

# **How Can You Change the Default Port in Spring Boot?**

We can [change the default port of a server embedded in Spring Boot](https://www.baeldung.com/spring-boot-change-port) using one of these ways:

* using a properties file – we can define this in an application.properties (or application.yml) file using the property server.port
* programmatically – in our main @SpringBootApplication class, we can set the server.port on the SpringApplication instance
* using the command line – when running the application as a jar file, we can set the server.port as a java command argument:

java -jar -Dserver.port=8081 myspringproject.jar

The default port number of the tomcat server to run the spring boot application is 8080, which is further possible to change it.

So we can change the port of tomcat following ways given below:-

* Using application.properties
* Using application.yml
* Using EmbeddedServletContainerCustomizer interface.
* Using WebServerFactoryCustomizer interface.
* Using Command-Line Parameter.

1. Command-line:

https://lh5.googleusercontent.com/Ma9p-THLWn5Gym5YVECrbNyuOCv2pcNo5L3Xm6KRc9ttrFrk_sM3qLv491JNIMScY98neGKqX9HK4YKfoKBdEOYrQ95LYwbXShzpRDxqAa344c9Au9ySJS_b1bluRNcOFYSEke4

1. By changing in application.properties file

https://lh4.googleusercontent.com/KDpPC-2_6ADgzJZybqLPHBFVa4TazZOofy3R4z2ETRvR85P_buEEPVO1PJEZAtiFlPX8ZdL1IgfJL9ZfOu-tmzwaOO3fRiX9siWjal-OCSTvMPsx2xZDODIGaMMnQBQSZAIC--Y

1. Programmatic Configuration:



# **How to get the list of all the beans in your Spring boot application?**

Spring Boot actuator “/Beans” is used to get the list of all the spring beans in your application.

In the case of spring boot, you can use appContext.getBeanDefinitionNames() to get all the beans loaded by the spring container.By calling this method, we can show all of our beans present in our spring boot applications.

# **Can you control logging with Spring Boot? How?**

Yes, we can control logging with Spring Boot by specifying log levels on application.properties file. Spring Boot loads this file when it exists in the classpath and it can be used to configure both Spring Boot and application code.

Spring Boot uses Commons Logging for all internal logging and you can change log levels by adding following lines in the application.properties file:

logging.level.org.springframework=DEBUG

logging.level.com.demo=INFO

# **What is Spring Boot Devtools Used For?**

Spring Boot Developer Tools, or DevTools, is a set of tools making the development process easier. To include these development-time features, we just need to add a dependency to the pom.xml file:

<**dependency**>

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

<**artifactId**>spring-boot-devtools</**artifactId**>

</**dependency**>

The spring-boot-devtools module is automatically disabled if the application runs in production. The repackaging of archives also excludes this module by default. Hence, it won't bring any overhead to our final product.

By default, DevTools applies properties suitable to a development environment. These properties disable template caching, enable debug logging for the web group, and so on. As a result, we have this sensible development-time configuration without setting any properties.

**Applications using DevTools restart whenever a file on the classpath changes.** This is a very helpful feature in development, as it gives quick feedback for modifications.

By default, static resources, including view templates, don't set off a restart. Instead, a resource change triggers a browser refresh. Notice this can only happen if the LiveReload extension is installed in the browser to interact with the embedded LiveReload server that DevTools contains.

# **What is LiveReload in Spring Boot**

LiveReload is a spring-boot-devtools module that includes LiveReload server to trigger a browser refresh when a resource is changed.

LiveReload server extensions are available freeware for Firefox, Chrome, and Safari.

# **How to resolve whitelabel error page in spring boot application?**

This is quite common error in spring boot application which says 404(page not found).

We can mostly resolve this in 3 ways:

1. **Custom Error Controller**– where you will be implementing ErrorController interface which is provided by SpringFramework and then overriding its getErrorPath() so that you can return a custom path whenever such type of error is occurred.
2. **By Displaying Custom error page**– All you have to do is create an error.html page and place it into the src/main/resources/templates path. The BasicErrorController of of springboot will automatically pick this file by default.
3. **By disabling the whitelabel error page**– this is the easiest way where all you need to do is server.error.whitelabel.enabled property to false in the application.properties file to disable the whitelabel error page.

# ResponseEntity

ResponseEntity **represents the whole HTTP response: status code, headers, and body**. As a result, we can use it to fully configure the HTTP response.

If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

ResponseEntity is a generic type. Consequently, we can use any type as the response bod

@GetMapping("/hello")

ResponseEntity<String> **hello**() {

**return** **new** ResponseEntity<>("Hello World!", HttpStatus.OK);

}

Since we specify the response status programmatically, we can return with different status codes for different scenarios:

@GetMapping("/age")

ResponseEntity<String> **age**(

@RequestParam("yearOfBirth") **int** yearOfBirth) {

**if** (isInFuture(yearOfBirth)) {

**return** **new** ResponseEntity<>(

"Year of birth cannot be in the future",

HttpStatus.BAD\_REQUEST);

}

**return** **new** ResponseEntity<>(

"Your age is " + calculateAge(yearOfBirth),

HttpStatus.OK);

}

Additionally, we can set HTTP headers:

@GetMapping("/customHeader")

ResponseEntity<String> **customHeader**() {

HttpHeaders headers = **new** HttpHeaders();

headers.add("Custom-Header", "foo");

**return** **new** ResponseEntity<>(

"Custom header set", headers, HttpStatus.OK);

}

Furthermore, ResponseEntity **provides two nested builder interfaces**: HeadersBuilder and its subinterface, BodyBuilder. Therefore, we can access their capabilities through the static methods of ResponseEntity.

The simplest case is a response with a body and HTTP 200 response code:

@GetMapping("/hello")

ResponseEntity<String> **hello**() {

**return** ResponseEntity.ok("Hello World!");

}

For the most popular [HTTP status codes](https://www.baeldung.com/cs/http-status-codes) we get static methods:

BodyBuilder **accepted**();

BodyBuilder **badRequest**();

BodyBuilder **created**(java.net.URI location);

HeadersBuilder<?> noContent();

HeadersBuilder<?> notFound();

BodyBuilder **ok**();

In addition, we can use the BodyBuilder status(HttpStatus status) and the BodyBuilder status(int status) methods to set any HTTP status.

Finally, with ResponseEntity<T> BodyBuilder.body(T body) we can set the HTTP response body:

@GetMapping("/age")

ResponseEntity<String> **age**(@RequestParam("yearOfBirth") **int** yearOfBirth) {

**if** (isInFuture(yearOfBirth)) {

**return** ResponseEntity.badRequest()

.body("Year of birth cannot be in the future");

}

**return** ResponseEntity.status(HttpStatus.OK)

.body("Your age is " + calculateAge(yearOfBirth));

}

We can also set custom headers:

@GetMapping("/customHeader")

ResponseEntity<String> **customHeader**() {

**return** ResponseEntity.ok()

.header("Custom-Header", "foo")

.body("Custom header set");

}

Since BodyBuilder.body() returns a ResponseEntity instead of BodyBuilder, it should be the last call.

Note that with HeaderBuilder we can't set any properties of the response body.

While returning ResponseEntity<T> object from the controller, we might get an exception or error while processing the request and would like to **return error-related information to the user represented as some other type, let's say E**.

Spring 3.2 brings support for a global **@ExceptionHandler with the new @ControllerAdvice annotation,** which handles these kinds of scenarios. For in-depth details, refer to our existing article [here](https://www.baeldung.com/exception-handling-for-rest-with-spring).

**While ResponseEntity is very powerful, we shouldn't overuse it.** In simple cases, there are other options that satisfy our needs and they result in much cleaner code.

## 3. Alternatives

### 3.1. @ResponseBody

In classic Spring MVC applications, endpoints usually return rendered HTML pages. Sometimes we only need to return the actual data; for example, when we use the endpoint with AJAX.

In such cases, we can mark the request handler method with @ResponseBody, and **Spring treats the result value of the method as the HTTP response body** itself.

For more information, [this article is a good place to start](https://www.baeldung.com/spring-request-response-body).

### 3.2. @ResponseStatus

When an endpoint returns successfully, Spring provides an HTTP 200 (OK) response. If the endpoint throws an exception, Spring looks for an exception handler that tells which HTTP status to use.

We can mark these methods with @ResponseStatus, and therefore, Spring **returns with a custom HTTP status**.

For more examples, please visit our article about [custom status codes](https://www.baeldung.com/spring-response-status).

### 3.3. Manipulate the Response Directly

Spring also lets us access the javax.servlet.http.HttpServletResponse object directly; we only have to declare it as a method argument:

@GetMapping("/manual")

**void** **manual**(HttpServletResponse response) **throws** IOException {

response.setHeader("Custom-Header", "foo");

response.setStatus(200);

response.getWriter().println("Hello World!");

}

Since Spring provides abstractions and additional capabilities above the underlying implementation, **we shouldn't manipulate the response this way**.

# **How to exclude any package without using the basePackages filter?**

There are different ways you can filter any package. But Spring Boot provides a trickier option for achieving this without touching the component scan. You can use the exclude attribute while using the annotation @SpringBootApplication. See the following code snippet:

@SpringBootApplication(exclude= {Employee.class})

public class FooAppConfiguration {}

# **What is a shutdown in the actuator?**

[Shutdown](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html) is an endpoint that allows the application to be gracefully shutdown. This feature is not enabled by default. You can enable this by using management.endpoint.shutdown.enabled=true in your application.properties file. But be careful about this if you are using this.

# Name two ways to create a new Spring Boot project from scratch? Also, how do you know what spring-boot-starters your project needs?

You can create new Spring Boot projects through the [Spring Initializr](https://start.spring.io/) web application or the [Spring Boot CLI](https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started-installing-the-cli). Interestingly enough, Spring Initializr is not just a website where you can generate project skeleton .zip files. It is also an API, that you can programmatically call. All major IDEs (Spring Tool Suite, IntelliJ IDEA Ultimate, Netbeans and VSCode) directly integrate with it, so that you can create new Spring Boot projects right out of your IDE.

# **Why do you not need to specify dependency versions in your pom.xml file when including 3rd party libraries? Does that hold true for all 3rd party libraries or only some? How can you find out what libraries Spring Boot supports?**

This is because Spring Boot does some dependency management for you.

On a high-level, Spring Boot starters pull-in a parent pom.xml file (or a build.gradle file) which has all the dependencies and respective versions defined that a specific Spring Boot version supports - a so-called bill of materials. You can then simply use those pre-defined versions, or override the version numbers in your own build scripts.

You can find the list of all currently supported 3rd party libraries and versions in the [spring-boot-dependencies](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-dependencies/build.gradle) project.

# **What is the easiest way to deploy a Spring Boot application in production? What other options are there?**

The simplest way to deploy your Spring Boot application is as a .jar file with an embedded servlet container, to any server or platform that has a JRE installed.  
For organizational and historical reasons, you can also deploy your Spring Boot application as a .war file, into an existing servlet container or application server.

Last but not least, you can, of course, also put your .jar file into a Docker image and even deploy those with Kubernetes.

# **How to handle exceptions in spring boot?**

To handle exceptions in spring boot, you can use **@ControllerAdvice** annotation to handle your exceptions globally.

In order to handle specific exception and send customized response, you need to use **@ExceptionHandler** annotation.

@ControllerAdvice constructor comes with some special arguments, which allows you to scan only the related portion of your application and handle only those exceptions thrown by the respective classes mentioned in the constructor. By default, it will scan and handle all the classes in your application. Below are some types which we can use to restrict only specific classes to handle exceptions.

1) annotations - Controllers that are annotated with the mentioned annotations will be assisted by the @ControllerAdvice annotated class and are eligible for exception of those classes

eg. @ControllerAdvice(annotations = RestController.class) - Here the exception helper annotated by @ControllerAdvice will catch all the exceptions thrown by the @RestController annotation classes.

2) basePackages - By Specifying the packages that we want to scan and handling exceptions for the same.

eg. @ControllerAdvice(basePackages = "org.example.controllers") - This will only scan call the mentioned package and handle the exceptions for the same.

3) assignableTypes - This argument will make sure to scan and handle the exceptions from the mentioned classes

eg. @ControllerAdvice(assignableTypes = {ControllerInterface.class,   
AbstractController.class})

### **Before Using @ControllerAdvice**

In the below code snippet, we see there are many duplications of lines, and the controller code is not easily readable because of multiple try and catch blocks in each API.

**1**

**@RestController**

**2**

**@RequestMapping(path = "/employees")**

**3**

**public class EmployeeController {**

**4**

**5**

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

**6**

**7**

**private EmployeeDao employeeDao;**

**8**

**9**

**@GetMapping(path="/{employeeId}", produces = "application/json")**

**10**

**public ResponseEntity<Employee> getEmployees(@PathVariable Long employeeId) {**

**11**

**ResponseEntity<Employee> response = null;**

**12**

**try {**

**13**

**if(null==employeeId || positionId.equals(0L)) {**

**14**

**throw new InvalidInputException("Employee Id is not valid");**

**15**

**}**

**16**

**employee = employeeDao.getEmployeeDetails(employeeId);**

**17**

**response = new ResponseEntity<Employee>(employee,HttpStatus.OK);**

**18**

**}**

**19**

**catch(InvalidInputException e) {**

**20**

**Logger.error("Invalid Input:",e.getMessage());**

**21**

**response = new ResponseEntity<Employee>(employee,HttpStatus.BAD\_REQUEST);**

**22**

**}**

**23**

**catch(BusinessException e) {**

**24**

**Logger.error("Business Exception:",e.getMessage());**

**25**

**response = new ResponseEntity<Employee>(employee,HttpStatus.INTERNAL\_SERVER\_ERROR);**

**26**

**}**

**27**

**catch(Exception e) {**

**28**

**Logger.error("System Error:",e.getMessage());**

**29**

**response = new ResponseEntity<Employee>(employee,HttpStatus.INTERNAL\_SERVER\_ERROR);**

**30**

**}**

**31**

**return response;**

**32**

**}**

**33**

**34**

**@GetMapping(path="/address/{employeeId}", produces = "application/json")**

**35**

**public ResponseEntity<Address> getEmployeeAddress(@PathVariable Long employeeId,@RequestHeader Long userId) {**

**36**

**ResponseEntity<Address> response = null;**

**37**

**try {**

**38**

**if(null==employeeId || positionId.equals(0L)) {**

**39**

**throw new InvalidInputException("Employee Id is not valid");**

**40**

**}**

**41**

**if(null==userId || userId.equals(0L)) {**

**42**

**throw new UnauthorizedException("Unauthorized user");**

**43**

**}**

**44**

**address = employeeDao.getEmployeeAddress(employeeId);**

**45**

**response = new ResponseEntity<Address>(address,HttpStatus.OK);**

**46**

**}**

**47**

**catch(UnauthorizedException e) {**

**48**

**Logger.error("Unauthorized:",e.getMessage());**

**49**

**response = new ResponseEntity<Address>(address,HttpStatus.BAD\_REQUEST);**

**50**

**}**

**51**

**catch(InvalidInputException e) {**

**52**

**Logger.error("Invalid Input:",e.getMessage());**

**53**

**response = new ResponseEntity<Address>(address,HttpStatus.BAD\_REQUEST);**

**54**

**}**

**55**

**catch(Exception e) {**

**56**

**Logger.error("System Error:",e.getMessage());**

**57**

**response = new ResponseEntity<Address>(address,HttpStatus.INTERNAL\_SERVER\_ERROR);**

**58**

**}**

**59**

**return response;**

**60**

**}**

**61**

**}**

**After Using @ControllerAdvice**

The below code snippet makes the code easily readable and also reduces duplications of lines.

**@RestController**

**@RequestMapping(path = "/employees")**

**public class EmployeeController {**

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

**@GetMapping(path="/{employeeId}", produces = "application/json")**

**public ResponseEntity<Employee> getEmployees(@PathVariable Long employeeId) {**

**if(null==employeeId || positionId.equals(0L)) {**

**throw new InvalidInputException("Employee Id is not valid");**

**11**

**}**

**12**

**Employee employee = employeeDao.getEmployeeDetails(employeeId);**

**13**

**return new ResponseEntity<Employee>(employee,HttpStatus.OK);;**

**14**

**}**

**15**

**16**

**@GetMapping(path="/address/{employeeId}", produces = "application/json")**

**17**

**public ResponseEntity<Address> getEmployeeAddress(@PathVariable Long employeeId,@RequestHeader Long userId) {**

**18**

**if(null==employeeId || employeeId.equals(0L)) {**

**19**

**throw new InvalidInputException("Employee Id is not valid");**

**20**

**}**

**21**

**if(null==userId || userId.equals(0L)) {**

**22**

**throw new UnauthorizedException("Unauthorized user");**

**23**

**}**

**24**

**Address address = employeeDao.getEmployeeAddress(employeeId,userId);**

**25**

**return new ResponseEntity<Address>(address,HttpStatus.OK);**

**26**

**}**

**27**

**}**

**28**

**29**

**30**

**@ControllerAdvice**

**31**

**public class ExceptionHelper {**

**32**

**33**

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

**34**

**35**

**@ExceptionHandler(value = { InvalidInputException.class })**

**36**

**public ResponseEntity<Object> handleInvalidInputException(InvalidInputException ex) {**

**37**

**LOGGER.error("Invalid Input Exception: ",ex.getMessage());**

**38**

**return new ResponseEntity<Object>(ex.getMessage(),HttpStatus.BAD\_REQUEST);**

**39**

**}**

**40**

**41**

**@ExceptionHandler(value = { Unauthorized.class })**

**42**

**public ResponseEntity<Object> handleUnauthorizedException(Unauthorized ex) {**

**43**

**LOGGER.error("Unauthorized Exception: ",ex.getMessage());**

**44**

**return new ResponseEntity<Object>(ex.getMessage(),HttpStatus.BAD\_REQUEST);**

**45**

**}**

**46**

**47**

**@ExceptionHandler(value = { BusinessException.class })**

**48**

**public ResponseEntity<Object> handleBusinessException(BusinessException ex) {**

**49**

**LOGGER.error("Business Exception: ",ex.getMessage());**

**50**

**return new ResponseEntity<Object>(ex.getMessage(),HttpStatus.INTERNAL\_SERVER\_ERROR);**

**51**

**}**

**52**

**53**

**@ExceptionHandler(value = { Exception.class })**

**54**

**public ResponseEntity<Object> handleException(Exception ex) {**

**55**

**LOGGER.error("Exception: ",ex.getMessage());**

**56**

**return new ResponseEntity<Object>(ex.getMessage(),HttpStatus.INTERNAL\_SERVER\_ERROR);**

**57**

**}**

**58**

**}**

# **Which is the ui web framework that is built to use spring boot?**

The best UI web framework that can be used with springboot is **JHipster.**

With this you can generate your web-applications and microservices within less time.

# **How can we create a custom endpoint in Spring Boot Actuator?**

By using @Endpoint annotation, you can create a custom endpoint.

# **How to handle 404 error in spring boot?**

Consider a scenario, where there are no stockDetails in the DB and still, whenever you hit the GET method you get 200(OK) even though the resource is not found which is not expected. Instead of 200, you should get 404 error.  
So to handle this, you need to create an exception, in the above scenario “StockNotFoundException”.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **GetMapping("/stocks/{number}")**  **public Stock retriveStock(@PathVariable int number)**  **{**  **Stock stock = service.findOne(number);**  **if(Stock ==null)**  **//runtime exception**  **throw new StockNotFoundException("number: "+ number);**  **return stock;**  **}** |

Now, create a Constructor from Superclass.

Right-click on the file -> Go to Source ->And generate constuctors from superclass-> and check the RuntimeException(String)-> and generate.

# **And add an annotation called @ResponseStatus which will give you 404(not found) error.**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **package com.greatlearning;**  **import org.springframework.http.HttpStatus;**  **import org.springframework.web.bind.annotation.ResponseStatus;**    **@ResponseStatus(HttpStatus.NOT\_FOUND)**  **public class StockNotFoundException extends RuntimeException**  **{**  **public StockNotFoundException(String message)**  **{**  **super(message);**  **}**  **}** |

Now, you can hit the same URL again and there you go, you get a 404 error when a resource is not found.

# **How to do pagination in spring boot?**

The process of dividing your data into small and suitable chunks is Pagination.

One can achieve pagination by using PagingAndSortingRepository which is an extension of crudRepository.

# **How to enable debugging log in the spring boot application?**

The user can enable a “debug” mode in the spring boot application by starting your application with a –debug flag. The user can also specify debug=true in the application.properties

When the debug mode is enabled, a user can configure a selection of core loggers (embedded container, Hibernate, and Spring Boot) to output more information.

# **Where do we define properties in the Spring Boot application?**

**Command Line Properties**

Command-line properties are converted into Spring Boot Environment properties by the spring boot application.

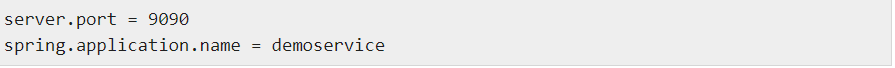
Command-line properties have more precedence over the other property sources.

Spring Boot uses the 8080 port number, by default, to start the Tomcat. Let us see how one can change the port number by using command-line properties.



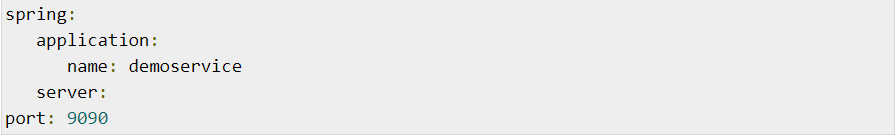
**Properties File**

Properties files are used to keep one or more properties in a single file to run the application in a different environment. Properties are kept in the application.properties file under the classpath in a typical spring boot application. The location of the application.properties file is at src/main/resources directory. The code of application.properties file is as below:



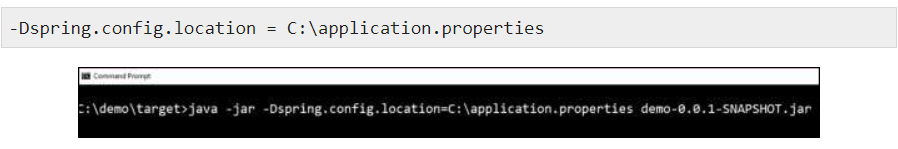
**YAML File**

Spring Boot also supports YAML-based properties configurations to run the application. The user can use, application.yml file instead of the application.properties file. The YAML file is kept inside the classpath. The sample application.yml file is given below −



**Externalized Properties**

The user can keep properties in different locations or paths instead of keeping the properties file under classpath. While running the JAR file, the user can specify the properties file path. The application developer can use the following command to specify the location of the properties file while running the JAR −



# **What is Open API?**

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the servicewithout access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools,and many other use cases.

OpenAPI Document

A document (or set of documents) that defines or describes an API. An OpenAPI definition uses and conforms tothe OpenAPI Specification.An OpenAPI document that conforms to the OpenAPI Specification is itself a JSON object, which may be represented either in JSON or YAML format.

# **How to document your Spring Boot REST APIs using Swagger with SpringFox?**

Swagger™ is a project used to describe and document RESTful APIs.With Swagger UI it is possible to explore our REST API with a nice graphical user interface in our browser.Using Swagger makes documenting your RESTful services easy.

Swagger 2 is an open-source project used to describe and document RESTful APIs. Swagger 2 is language-agnostic and is extensible into new technologies and protocols beyond HTTP. The current version defines a set HTML, JavaScript, and CSS assets to dynamically generate documentation from a Swagger-compliant API. These files are bundled by the Swagger UI project to display the API on the browser. Besides rendering documentation, Swagger UI allows other API developers or consumers to interact with the API’s resources without having any of the implementation logic in place.

The Swagger 2 specification, which is known as OpenAPI specification, has several implementations. Currently, Springfox that has replaced Swagger-SpringMVC (Swagger 1.2 and older) is popular for Spring Boot applications.

Springfox supports both Swagger 1.2 and 2.0.

Adding Dependencies To work with SpringFox in your project, you need to add it as a dependency first. If you are using Maven, you can use the following (you can check whether a newer version is available).

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

@Configuration

@EnableSwagger2

public class SpringFoxConfig {

@Bean

public Docket apiDocket() {

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.any())

.paths(PathSelectors.any())

.build();

}

}

Adding Swagger Core annotations to your controller classes

Same as you could annotate your model classes with Swagger core annotations to provide additional metadata,

you can annotate your controllers and their methods and method parameters.

**@Api describes the whole controller**

**@ApiOperation is used for description on a methods level**

**@ApiParam is used for method parameters**

**@RestController**

**@RequestMapping("/v2/persons/")**

**@Api(description = "Set of endpoints for Creating, Retrieving, Updating and Deleting of Persons.")**

**public class PersonController {**

**private PersonService personService;**

**@RequestMapping(method = RequestMethod.GET, produces = "application/json")**

**@ApiOperation("Returns list of all Persons in the system.")**

**public List getAllPersons() {**

**return personService.getAllPersons();**

**}**

**@RequestMapping(method = RequestMethod.GET, path = "/{id}", produces = "application/json")**

**@ApiOperation("Returns a specific person by their identifier. 404 if does not exist.")**

**public Person getPersonById(@ApiParam("Id of the person to be obtained. Cannot be empty.")**

**@PathVariable int id) {**

**return personService.getPersonById(id);**

**}**

**@RequestMapping(method = RequestMethod.DELETE, path = "/{id}")**

**@ApiOperation("Deletes a person from the system. 404 if the person's identifier is not found.")**

**public void deletePerson(@ApiParam("Id of the person to be deleted. Cannot be empty.")**

**@PathVariable int id) {**

**personService.deletePerson(id);**

**}**

**@RequestMapping(method = RequestMethod.POST, produces = "application/json")**

**@ApiOperation("Creates a new person.")**

**public Person createPerson(@ApiParam("Person information for a new person to be created.")**

**@RequestBody Person person) {**

**return personService.createPerson(person);**

**}**

**@Autowired**

**public void setPersonService(PersonService personService) {**

**this.personService = personService;**

**}**

**}**

In addition to Sprinfox, we also require Swagger UI. The code to include Swagger UI is this.

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.6.1</version>

<scope>compile</scope>

</dependency>