<https://dzone.com/articles/top-10-spring-boot-interview-questions>

## ****1) What does the @SpringBootApplication annotation do internally?****

As per the Spring Boot doc, the [@SpringBootApplication](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html) annotation is equivalent to using [@Configuration](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html), [@EnableAutoConfiguration](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html), and [@ComponentScan](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) 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 individual annotation as per our project needs.

## ****2) 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 {}

**3) How to disable a specific auto-configuration class?**

You can use the exclude attribute of@EnableAutoConfiguration, if you find any specific auto-configuration classes that you do not want are being applied.

//By using "exclude"

@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

On the other foot, if the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead.

//By using "excludeName"

@EnableAutoConfiguration(excludeName={Foo.class})

Also, Spring Boot provides the facility to control the list of auto-configuration classes to exclude by using the spring.autoconfigure.exclude property. You can add into the application.properties. And you can add multiple classes with comma separated.

//By using property file

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

## ****4) What is Spring Actuator? What are its advantages?****

This is one of the most common interview questions in Spring Boot. As per the Spring doc:

**"An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change."**

As we know, Spring Boot provides lots of auto-configuration features that help developers quickly develop production components. But if you think about debugging and how to debug, if something goes wrong, we always need to analyze the logs and dig through the data flow of our application to check to see what's going on. So, the Spring Actuator provides easy access to those kinds of features. It provides many features, i.e. what beans are created, the mapping in the controller, the CPU usage, etc. Automatically gathering and auditing health and metrics can then be applied to your application.

It provides a very easy way to access the few production-ready [REST endpoints](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html) and fetch all kinds of information from the web. But by using these endpoints, you can do many things to see here the endpoint docs. There is no need to worry about security; if Spring Security is present, then these endpoints are secured by default using Spring Security’s content-negotiation strategy. Or else, we can configure custom security by the help of RequestMatcher.

## ****5) How to enable/disable the Actuator?****

Enabling/disabling the actuator is easy; the simplest way is to enable features to add the dependency (Maven/Gradle) to the spring-boot-starter-actuator, i.e. Starter. If you don't want the actuator to be enabled, then don't add the dependency.

**Maven dependency:**

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

## ****6) What is the Spring Initializer?****

This may not be a difficult question, but the interviewer always checks the subject knowledge of the candidate. It's often that you can't always expect questions that you have prepared. However, this is a very common question asked almost all of the time.

The Spring Initializer is a web application that generates a Spring Boot project with everything you need to start it quickly. As always, we need a good skeleton of the project; it helps you to create a project structure/skeleton properly. You can learn more about the [Initializer here](https://start.spring.io/).

## ****7) 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.

## ****8) Is this possible to change the port of Embedded Tomcat server in Spring boot?****

Yes, it's possible to change the port. You can use the application.properties file to change the port. But you need to mention "server.port" (i.e. server.port=8081). Make sure you have application.properties in your project classpath; REST Spring framework will take care of the rest. If you mention server.port=0 , then it will automatically assign any available port.

## ****9) Can we override or replace the Embedded Tomcat server in Spring Boot?****

Yes, we can replace the Embedded Tomcat with any other servers by using the Starter dependencies. You can use spring-boot-starter-jetty  or spring-boot-starter-undertow as a dependency for each project as you need.

## ****10) Can we disable the default web server in the Spring Boot application?****

The major strong point in Spring is to provide flexibility to build your application loosely coupled. Spring provides features to disable the web server in a quick configuration. Yes, we can use the application.properties to configure the web application type, i.e.  spring.main.web-application-type=none.

**Enabling Spring Boot Actuator:**

https://www.baeldung.com/spring-boot-actuator-enable-endpoints

To enable spring boot actuator endpoints we need to add following in application.properties file.

management.endpoints.web.exposure.include=\*

The management.endpoints.web.exposure.include property can also take a comma-separated list of endpoints. So, let's only expose /beans and /loggers:

management.endpoints.web.exposure.include=beans, loggers

In addition to including certain endpoints with a property, we can also exclude endpoints. Let's expose all the endpoints except /threaddump:

management.endpoints.web.exposure.include=\*

management.endpoints.web.exposure.exclude=threaddump

Both the include and exclude properties take a list of endpoints. **The exclude property takes precedence over include**.

https://www.mygreatlearning.com/blog/spring-boot-interview-questions/

### ****5. What is actuator in spring boot?****

An actuator is one of the best parts of spring boot which consists of production-ready features to help you monitor and manage your application.

With the help of an actuator, you can monitor what is happening inside the running application.  
Actuator dependency figures out the metrics and makes them available as a new endpoint in your application and retrieves all required information from the web. You can identify beans, the health status of your application, CPU usage, and many more with the actuator.

### ****11. What is auto configuration in spring boot?****

AutoConfiguration is a process by which Spring Boot automatically configures all the infrastructural beans. It declares the built-in beans/objects of the spring specific module such as JPA, spring security and so on based on the dependencies present in your applications class path.

**For example:** If we make use of Spring JDBC, the spring boot autoconfiguration feature automatically registers the DataSource and JDBCTemplete bean.  
This entire process of automatically declaring the framework specific bean without the need of writing the xml code or java config code explicity  is called Autoconfiguration which is done by springBoot with the help of an annotation called **@EnableAutoconfiguration** alternatively **@SpringBootApplication**.

### ****17. What is bootstrapping in spring boot?****

One of the way to bootstrap your spring boot application is using Spring Initializer.  
you can go to the official website of spring  and select your version, and add you groupID, artifactId and all the required dependencies.

And then you can create your restEndpoints and build and run your project.  
There you go, you have bootstrapped your spring boot application.

### ****34) What do you understand  by auto-configuration in Spring Boot and how to disable the auto-configuration?****

AutoConfiguration is a process by which Spring Boot automatically configures all the infrastructural beans. It declares the built-in beans/objects of the spring-specific module such as JPA, spring-security, and so on based on the dependencies present in your application’s classpath.  
**For example:** If we make use of Spring JDBC, the spring boot autoconfiguration feature automatically registers the DataSource and JDBCTemplete bean.  
This entire process of automatically declaring the framework-specific bean without the need of writing the XML code or java-config code explicitly  is called Autoconfiguration which is done by spring-boot with the help of an annotation called **@EnableAutoconfiguration** alternatively **@SpringBootApplication.**

1. You can exclude the attribute of @EnableAutoConfiguration where you don’t want it to be configured implicity in order to disable the spring boot’s auto-configuration feature.

2. Another way of disabling auto-configuration is by using the property file:

spring.autoconfigure.exclude=

org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,

org.springframework.boot.autoconfigure.data.MongoDataConfiguration,

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

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

<https://www.baeldung.com/spring-boot-actuators>

## ****What Is an Actuator?****

In essence, Actuator brings production-ready features to our application.

**Monitoring our app, gathering metrics, understanding traffic, or the state of our database become trivial with this dependency.**

The main benefit of this library is that we can get production-grade tools without having to actually implement these features ourselves.

Actuator is mainly used to **expose operational information about the running application** — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

Once this dependency is on the classpath, several endpoints are available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

### ****2.1. Getting Started****

To enable Spring Boot Actuator, we just need to add the spring-boot-actuator dependency to our package manager.

In Maven:

<**dependency**>

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

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

</**dependency**>

Note that this remains valid regardless of the Boot version, as versions are specified in the Spring Boot Bill of Materials (BOM).

## ****3. Spring Boot 2.x Actuator****

In 2.x, Actuator keeps its fundamental intent but simplifies its model, extends its capabilities, and incorporates better defaults.

First, this version becomes technology-agnostic. It also simplifies its security model by merging it with the application one.

Among the various changes, it's important to keep in mind that some of them are breaking. This includes HTTP requests and responses as well as Java APIs.

Lastly, the latest version now supports the CRUD model as opposed to the old read/write model.

### ****3.1. Technology Support****

With its second major version, Actuator is now technology-agnostic whereas in 1.x it was tied to MVC, therefore to the Servlet API.

In 2.x, Actuator defines its model as pluggable and extensible without relying on MVC for this.

**Hence, with this new model, we're able to take advantage of MVC as well as WebFlux as an underlying web technology.**

Moreover, forthcoming technologies could be added by implementing the right adapters.

Finally, JMX remains supported to expose endpoints without any additional code.

### ****3.2. Important Changes****

Unlike in previous versions, **Actuator comes with most endpoints disabled.**

Thus, the only two available by default are /health and /info.

If we want to enable all of them, we could set management.endpoints.web.exposure.include=\*. Alternatively, we can list endpoints that should be enabled.

**Actuator now shares the security config with the regular App security rules, so the security model is dramatically simplified.**

Therefore, to tweak Actuator security rules, we could just add an entry for /actuator/\*\*:

@Bean **public** SecurityWebFilterChain **securityWebFilterChain**( ServerHttpSecurity http) { **return** http.authorizeExchange() .pathMatchers("/actuator/\*\*").permitAll() .anyExchange().authenticated() .and().build(); }

We can find further details on the [brand new Actuator official docs](https://docs.spring.io/spring-boot/docs/2.0.x/actuator-api/html/).

Also, **by default, all Actuator endpoints are now placed under the /actuator path.**

Same as in the previous version, we can tweak this path using the new property management.endpoints.web.base-path.

### ****3.3. Predefined Endpoints****

Let's have a look at some available endpoints, most of which were available in 1.x already.

Also, **some endpoints have been added, some removed and some have been restructured**:

* */auditevents*lists security audit-related events such as user login/logout. Also, we can filter by principal or type among other fields.
* */beans*returns all available beans in our *BeanFactory*. Unlike */auditevents*, it doesn't support filtering.
* */conditions*, formerly known as /*autoconfig*, builds a report of conditions around autoconfiguration.
* */configprops*allows us to fetch all *@ConfigurationProperties*beans.
* */env*returns the current environment properties. Additionally, we can retrieve single properties.
* */flyway*provides details about our Flyway database migrations.
* */health*summarizes the health status of our application.
* */heapdump*builds and returns a heap dump from the JVM used by our application.
* */info*returns general information. It might be custom data, build information or details about the latest commit.
* */liquibase*behaves like */flyway*but for Liquibase.
* */logfile*returns ordinary application logs.
* */loggers*enables us to query and modify the logging level of our application.
* */metrics*details metrics of our application. This might include generic metrics as well as custom ones.
* */prometheus*returns metrics like the previous one, but formatted to work with a Prometheus server.
* */scheduledtasks*provides details about every scheduled task within our application.
* */sessions*lists HTTP sessions given we are using Spring Session.
* */shutdown*performs a graceful shutdown of the application.
* */threaddump*dumps the thread information of the underlying JVM.