**Q1. What is Spring Boot and What Are Its Main Features?**

Spring Boot is a framework for rapid application development built on top of the Spring Framework. With its auto-configuration and embedded application server support.

Here are a few 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" \l "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 the application.
* **[Security](https://www.baeldung.com/security-spring)**
* **[Logging](https://www.baeldung.com/spring-boot-logging)**

**Q2. What Are the Differences Between Spring and Spring Boot?**

**Spring :**

> Spring Framework is a widely used Java EE framework for building applications. Spring is an open-source lightweight framework widely used to develop enterprise applications.

> It aims to simplify Java EE development that makes developers more productive.

> The primary feature of the Spring Framework is dependency injection.

> It helps to create a loosely coupled application.

> The developer writes a lot of code (boilerplate code) to do the minimal task.To create a Spring application.

> To run the Spring application, we need to set the server explicitly.

> It does not provide support for an in-memory database

> Developers manually define dependencies for the Spring project in pom.xml.To run the Spring application, a deployment descriptor is required.

**Spring Boot :**

> Spring Boot Framework is widely used to develop REST APIs.

> The primary feature of Spring Boot is Auto configuration. It automatically configures the classes based on the requirement.

> It helps to create a stand-alone application with less configuration.

> It reduces boilerplate code.It reduces the lines of code.

> Spring Boot offers embedded server such as Jetty and Tomcat, etc.

> It offers several plugins for working with an embedded and in-memory database such as H2.

> Spring Boot comes with the concept of starter in pom.xml file that internally takes care of downloading the dependencies JARs based on Spring Boot Requirement.

> Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.

> There is no requirement for a deployment descriptor.Web,xml

**Spring Boot** and **Spring MVC** exist for different purposes. The primary **comparison** between Spring Boot and Spring MVC are discussed below:

**Spring Boot**

> Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults.

> It provides default configurations to build Spring-powered framework.

> There is no need to build configuration manually.

> There is no requirement for a deployment descriptor. Web. xml, also known as deployment descriptor, is traditionally used as a configuration file for Java web applications. It defines servlets, their mappings, servlet filters, lifecycle listeners and more. ... With Servlet version 3.0, the deployment descriptor is no longer mandatory.

> It avoids boilerplate code and wraps dependencies together in a single unit.

> It reduces development time and increases productivity.

**Spring MVC**

> Spring MVC is a model view controller-based web framework under the Spring framework.

> It provides ready to use features for building a web application.

> It requires build configuration manually.

> A Deployment descriptor is required.

> It specifies each dependency separately.

> It takes more time for development time & increase productivity.

|  |  |
| --- | --- |
| **Spring** | **Spring boot** |
| the Spring framework focuses on providing flexibility through its dependency injection feature. It helps to inject the required dependencies quickly but also to develop your application in a loosely coupled fashion | **Autoconfiguration**: Developers can automatically configure their Spring application. However, Spring Boot is also capable of changing the configuration based on the dependencies you list. For example, when you list “MySQL” as a dependency, it will configure your Spring application with the “MySQL connector” included. And if you want to add a custom configuration, you can create a class that overrides the default configuration for your “MySQL connector”. |
| A lightweight framework. | **Standalone**: There’s no need to deploy your application to a web server. You simply enter the run command to start the application. |
| Helps with loose coupling dependencies and testability. The modular architecture allows you to pick the parts you need and isolate them. | **Opinionated**: On the [official page](https://spring.io/projects/spring-boot" \t "https://stackify.com/what-is-spring-boot/_blank), we find that Spring Boot decides for you which defaults to use for the configuration. Also, it decides which packages to install for the dependencies you require. For example, if you include the Spring Boot starter “pom” for “JPA”, it will auto configure an in-memory database, a hibernate entity manager, and a simple data source. This is an example of an opinionated default configuration that you can override. While some developers might feel this is too opinionated, Spring Boot’s opinionated setup helps developers to get started quickly on their projects. |
| Has support for both XML and annotation configuration. |  |

**Some additional benefits include :**

* Reduces development time and increases the overall productivity of the development team.
* Helps you autoconfigure all components for a production-grade Spring application.
* Makes it easier for developers to create and test Java-based applications by providing a default setup for unit and integration tests.
* Avoids writing lots of boilerplate code, annotations, and XML configuration.
* Comes with embedded HTTP servers like [Tomcat or Jetty](https://stackify.com/tomcat-vs-jetty-vs-glassfish-vs-wildfly/) to test web applications.
* Adds many plugins that developers can use to work with embedded and in-memory databases easily. Spring allows you to easily connect with database and queue services like Oracle, PostgreSQL, MySQL, MongoDB, Redis, Solr, ElasticSearch, Rabbit MQ, ActiveMQ, and [many more](https://spring.io/guides" \t "https://stackify.com/what-is-spring-boot/_blank).
* Allows admin support—meaning you can manage via remote access to the application.

## **Why Spring Boot? :** You can choose Spring Boot because of the features and benefits it offers as given here −

* It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
* It provides a powerful batch processing and manages REST endpoints.
* In Spring Boot, everything is auto configured; no manual configurations are needed.
* It offers annotation-based spring application
* Eases dependency management
* It includes Embedded Servlet Container.

**How Spring Boot Application Works Internally ?**

**Spring Boot Application Internal Working :** 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 or 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 {}*

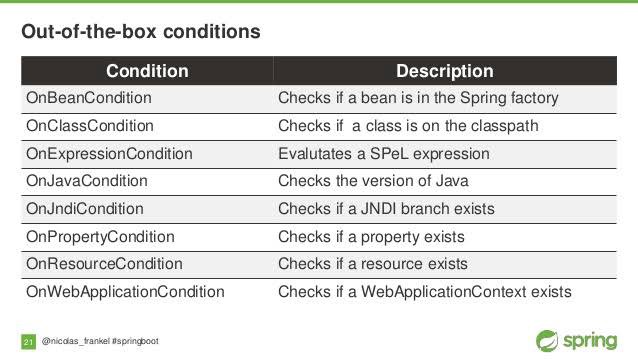
**@ConditionalOnBean(DataSource.class) :**— — — — — — — — — — — — — — —  
It will serach for the DataSource bean if it is available then only it will enable JpaRepositoriesAutoConfiguration . So this we need to define DataSource related properties into our property file.

**@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)

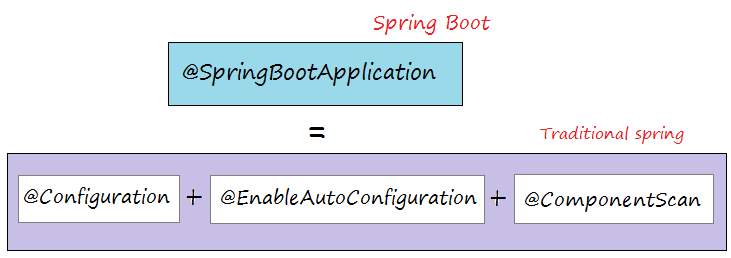
If all conditions are true then only it will enable JpaRepositoriesAutoConfiguration class.

**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…..  
}

**Application Bootstrap:** The basic **difference** in bootstrapping of an application in **Spring** and **Spring Boot** lies with the servlet. Spring uses either the **web.xml** or **SpringServletContainerInitializer** as its bootstrap entry point.

On the other hand, Spring Boot uses only Servlet 3 features to bootstrap an application.

**How Spring Bootstraps? (how spring works)?**

Spring supports both the legacy **web.xml** way of bootstrapping as well as the latest **Servlet 3+** method.

**Let's see the web.xml approach in steps:**

> Servlet container (the server) reads web.xml

> The DispatcherServlet defined in the web.xml is instantiated by the container

> DispatcherServlet creates WebApplicationContext by reading WEB-INF/{servletName}-servlet.xml

> Finally, the DispatcherServlet registers the beans defined in the application context.

**Here's how Spring bootstraps using Servlet 3+ approach:**

> The container searches for classes implementing ServletContainerInitializer and executes.

> The SpringServletContainerInitializer finds all classes implementing WebApplicationInitializer

> The WebApplicationInitializer creates the context with XML or @Configuration classes.

> The WebApplicationInitializer creates the DispatcherServlet with the previously created context.

**How Spring Boot Bootstraps? (how spring Boot works)?**

The entry point of a Spring Boot application is the class which is annotated with **@SpringBootApplication:**

**@SpringBootApplication**

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

By default, Spring Boot uses an embedded container to run the application. Spring Boot uses the **public static void main** entry-point to launch an embedded web server.

Also, it takes care of the binding of the Servlet, Filter, and ServletContextInitializer beans from the application context to the embedded servlet container.

Another feature of Spring Boot is that it automatically scans all the classes in the same package or sub packages of the Main-class for components.

Spring Boot provides the option of deploying it as a web archive in an external container as well. In this case, we have to extend the **SpringBootServletInitializer**:

**@SpringBootApplication**

public class Application extends **SpringBootServletInitializer** {

// ...

}

Here the external servlet container looks for the Main-class defined in the META-INF file of the web archive and the SpringBootServletInitializer will take care of binding the Servlet, Filter, and ServletContextInitializer.

**Q3.How Can We Set up a Spring Boot Application With Maven? :** We can include Spring Boot in a Maven project just like we would any other library. However, the best way is to inherit from the spring-boot-starter-parent project and declare dependencies to Spring Boot starters.

Inheriting the spring-boot-starter-parent project is straightforward – we only need to specify a parent element in pom.xml:

**<parent>**

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

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

**<version>2.4.0.RELEASE</version>**

**</parent>**

**Q4. What is Spring Initializr? :** Spring Initializr is a convenient way to create a Spring Boot project.

We can go to the Spring Initializr 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.

**Q5. What Spring Boot Starters Are Available out There?**

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

**Q6. How to Disable a Specific Auto-Configuration? :** If we want to disable a specific auto-configuration, we can indicate it using the **exclude attribute of the @EnableAutoConfiguration annotation**.

// 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

**Q7. 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.

**Q8. 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() { ... }

}

**Q9. 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.

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.

**Q10. 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.

**Q11. What Are Possible Sources of External Configuration?**

Spring Boot provides support for external configuration, allowing us to run the same application in various environments. We can use **properties files, YAML files, environment variables, system properties, and command-line option arguments** to specify configuration properties.

We can then gain access to those properties using the @Value annotation, a bound object via the @ConfigurationProperties annotation, or the Environment abstraction.

**Q13. 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.

**Q14. How to Write Integration Tests? :** When running integration tests for a Spring application, we must have an ApplicationContext.

To make our life easier, Spring Boot provides a special annotation for testing – @SpringBootTest. This annotation creates an ApplicationContext from configuration classes indicated by its classes attribute.

**Q15. What Is Spring Boot Actuator Used For? :** 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

In case the classes attribute isn't set, Spring Boot searches for the primary configuration class. The search starts from the package containing the test up until it finds a class annotated with @SpringBootApplication or @SpringBootConfiguration.

**Q16. Which Is a Better Way to Configure a Spring Boot Project – Using Properties or YAML?**

YAML offers many advantages over properties files, such as:

> More clarity and better readability.

> Perfect for hierarchical configuration data, which is also represented in a better, more readable format.

> Support for maps, lists, and scalar types.

> Can include several profiles in the same file (since Spring Boot 2.4.0, this is possible for properties files too).

> However, writing it can be a little difficult and error-prone due to its indentation rules.

**Q17. 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.

**Q18. How Can You Change the Default Port in Spring Boot?**

We can change the default port of a server embedded in Spring Boot 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

**Q19. Which Embedded Servers does Spring Boot Support, and How to Change the Default?**

As of date, Spring MVC supports Tomcat, Jetty, and Undertow. Tomcat is the default application server supported by Spring Boot's web starter.

Spring WebFlux supports Reactor Netty, Tomcat, Jetty, and Undertow with Reactor Netty as default.

In Spring MVC, to change the default, let's say to Jetty, we need to exclude Tomcat and include Jetty in the dependencies:

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

</dependency>

Similarly, to change the default in WebFlux to UnderTow, we need to exclude Reactor Netty and include UnderTow in the dependencies.

**Q20. Why Do We Need Spring Profiles?**

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.

**spring.profiles.active=dev**

To set profiles programmatically, we can also use the SpringApplication class:

SpringApplication.setAdditionalProfiles("dev");

To set profiles using Maven in Spring Boot, we can specify profile names under spring-boot-maven-plugin in pom.xml:

<plugins>

<plugin>

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

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

<configuration>

<profiles>

<profile>dev</profile>

</profiles>

</configuration>

</plugin>

...

</plugins>

and execute the Spring Boot-specific Maven goal:

**mvn spring-boot:run**

**Q: 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.**

**Q: 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.

**Q: 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.

**Q: What is a shutdown in the actuator?**

Shutdown 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.

**Q: 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.

**Q: What is Spring Actuator? What are its advantages?**

"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 be applied to your application.

It provides a very easy way to access the few production-ready REST endpoints 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.

**Q: 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 {}

**Q: Can Spring Boot also be used to create non-web applications?**

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.

**Q: Can you disable particular auto-configuration in spring boot? Explain how?**

Yes, we can do that by

Using the exclude attribute of **@EnableAutoConfiguration**

@Configuration

**@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**

public class CustomConfiguration {}

using the exclude attribute for @SpringBootApplication annotation

**@SpringBootApplication(exclude= DataSourceAutoConfiguration.class)**

public class CustomApplication {}

**Q: Describe spring-boot-starter-parent?**

It is a unique starter which adds jars to the classpath for easy Maven or Gradle dependency management.

**Q:What embedded containers are supported by Spring Boot?**

Spring Boot supports three embedded containers:

* Tomcat (used by default)
* Undertow
* Jetty

**Q: What is Thymeleaf in Spring Boot?**

It is a Java-based server-side template engine that offers elegant and natural templates for a web application.

**Q: Can you name and briefly explain all the spring boot** components/features?

**The main features of Spring Boot are :**

**1. Starter dependency :**There are many dependencies in the Spring framework, and this feature aggregates them together. Spring Boot comes packed with several starter dependencies to enhance productivity.

Few of the Spring Boot Starters are Test Starter, Web Starter, Mail Starter, and more. For instance, if we want to use JPA and Spring for database access, we can add this starter dependency in the project-spring-boot-starter-data-jpa.

**2. Auto-Configuration :**This feature scans the classpath and searches the libraries/Jars in the classpath to provide the necessary configuration to design and run the application.

For example, while developing an application with Spring Boot, if there is a Thymeleaf.jar present in the classpath, it automatically can align the Thymeleaf template resolver and other settings.

**3. Spring Initializer :**It is a web application that simplifies the process of the project set up by creating the initial project structure and build scripts. It increases productivity by reducing development time.

**4. Spring Actuator :** Actuators are incredibly significant for microservices as they enable deployment-ready features like auditing, health check-up, log information, etc. for running Spring boot applications. Actuators have built-in management endpoints that are secured by default.

For example:

/beans– it exhibits the entire list of all Spring beans in your application.

/health— it displays application health information by monitoring the production system

**5. Spring CLI :**This feature allows the developers to use Groovy for writing the Spring boot application, hence resulting in a more concise code.

**Q: What is the need for Spring Boot?**

While Spring offers developers an ideal environment to develop large applications, the amount of configuration and its complexity makes it challenging to do so.

Here is where Spring Boot comes to rescue. Its features like pre-built templates and auto-configuration allow developers to use existing spring functionalities with more ease, minimum effort, and maximum efficiency.

**The main advantages of Spring Boot are:**

> It reduces development and testing time.

> It uses JavaConfig instead of XML.

> It provides an opinionated development approach.

> It offers starter projects or defaults for agile development.

> No separate web server is required; hence there is no need to boot up Glassfish, Tomcat, or any other server.

> Reduce Developement, Testing time and efforts.

> Avoid lots of maven imports and the various version conflicts.

> Quick start to development by providing defaults.

> Requires less configuration-Since there is no web.xml file. Simply add classes annotated with@Configuration and then you can add methods annotated with@Bean, and Spring will automagically load up the object and manage it like it always has. You can even add @Autowired to the bean method to have Spring autowire in dependencies needed for the bean.

> Environment Based Configuration-Using these properties, you can pass into the application which environment you are using with:-Dspring.profiles.active={enviornment}. Spring will then load up the subsequent application properties file at (application-{environment}.properties) after loading up the main application properties file.

**Q: What is JavaConfig?**

Spring **JavaConfig** is a product of the Spring community that provides a pure-Java approach to configuring the Spring IoC Container. It helps avoid using XML configurations.

The advantages of JavaConfig are:

**Object-oriented configuration.** Because configurations are defined as classes in JavaConfig, users can take full advantage of object-oriented features in Java. One configuration class may subclass another, overriding its @Bean methods, etc.

**Reduced or eliminated XML configuration.** The benefits of externalized configuration based on the principles of dependency injection have been proven.

JavaConfig provides developers with a pure-Java approach to configuring the Spring container that is conceptually similar to XML configuration. It is technically possible to configure the container using only JavaConfig configuration classes, however in practice many have found it ideal to mix-and-match JavaConfig with XML.

**Type-safe and refactoring-friendly**. JavaConfig provides a type-safe approach to configuring the Spring container. Thanks to Java 5.0's support for generics, it is now possible to retrieve beans by type rather than by name, free of any casting or string-based lookups.

**Q: How to reload my changes on Spring Boot without having to restart server?**

This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers.

One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart server. This will eliminates the need for manually deploying the changes every time. Spring Boot doesnt have this feature when it has released its first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment. It also provides H2-database console for better testing the application. The following dependency is used

**<dependency>**

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

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

**<optional>true</optional>**

**</dependency>**

The DevTool dependency usage for autorestart and H2 DB console.

**Q: How to implement JWT authentication for Spring Boot Application?**

JWT stands for JSON Web Token. JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. The client will need to authenticate with the server using the credentials only once. During this time the server validates the credentials and returns the client a JSON Web Token(JWT). For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like username and password.

**Q:How to disable Actuator endpoint security in Spring Boot?**

By default all sensitive HTTP endpoints are secured such that only users that have an ACTUATOR role may access them. Security is enforced using the standard **HttpServletRequest.isUserInRole** method.

We can disable security using -

**management.security.enabled=false**

It is suggested to disable security only if the actuator endpoints are accessed behind firewall.

**Q: How to run Spring boot application to custom port? :**In order to run a spring boot application on a custom port you can specify the port in application.properties.

server.port=8090

**Q: What is YAML?**

YAML is a human-readable data serialization language. It is commonly used for configuration files.

Compared to properties file, YAML file is much more structured and less confusing in case we want to add complex properties in the configuration file. As can be seen YAML has hierarchical configuration data.

**Q: How to implement security for Spring boot application? :** For Implementing security for Spring Boot we use the **spring-boot-starter-security** dependency and have to add the Security config. It requires very little code. Config class will have to extend **WebSecurityConfigurerAdapter** and override its methods.

**Q: Have you integrated Spring Boot and ActiveMQ? :** For integrating Spring Boot and ActiveMQ we use the **spring-boot-starter-activemq** dependency

It requires very little configuration and no boilerplate code.

**Q: Have you integrated Spring Boot and Apache Kafka? :**For integrating Spring Boot and Apache Kafka we use the **spring-kafka** dependency.

**Q: How to implement Pagination and Sorting with Spring Boot?**

Using Spring Boot achieving pagination is very simple. Using the Spring Data-JPA this is achieved passing pageable org.springframework.data.domain.Pageable to the repository methods.

**Q: What is Swagger? Have you implemented it using Spring Boot?**

Swagger is widely used for visualizing APIs, and with Swagger UI it provides online sandbox for frontend developers.

Swagger is a tool, a specification and a complete framework implementation for producing the visual representation of RESTful Web Services. It enables documentation to be updated at the same pace as the server. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Thus Swagger removes the guesswork in calling the service.

**Q:What is Spring Batch? How do you implement it using Spring Boot?**

Spring Boot Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advanced technical services and features that will enable extremely high-volume and high performance batch jobs though optimization and partitioning techniques.Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

**Q: How to implement Exception Handling using Spring Boot?**

Spring provides a very useful way to handle exceptions using **ControllerAdvice**.

We will be implementing a **ControlerAdvice** class which will handle all exceptions thrown by the controller class.

**Q: What is caching? Have you used any caching framework with Spring Boot?**

A cache is an area of local memory that holds a copy of frequently accessed data that is otherwise expensive to get or compute. Have used Hazelcast for caching.

**Q: What is Apache Kafka? How to integrate it with Spring Boot?**

Apache Kafka is a distributed publish-subscribe messaging system. It is a scalable, fault-tolerant, publish-subscribe messaging system which enables us to build distributed applications. It is an Apache Top Level project. Kafka is suitable for both offline and online message consumption.

**Q: How can we monitor all the Spring Boot Microservices?**

Spring Boot provides actuator endpoints to monitor metrics of individual microservices. These endpoints are very helpful for getting information about applications like if they are up, if their components like database etc are working good. But a major drawback or difficulty about using actuator enpoints is that we have to individually hit the enpoints for applications to know their status or health. Imagine microservices involving 50 applications, the admin will have to hit the actuator endpoints of all 50 applications. To help us deal with this situation, we will be using open source project located at https://github.com/codecentric/spring-boot-admin.

Built on top of Spring Boot Actuator, it provides a web UI to enable us visualize the metrics of multiple applications.

**Q: Have you used any Spring Cloud Components with Spring Boot?**

Have used Spring Cloud components like Netflix Eureka for Service Registration,Ribbon for Load Balancing.

**Q: 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**

**Q: Can you name some common Spring Boot Starter POMs?**

Some of the most common Spring Boot Start dependencies or POMs are spring-boot-starter, spring-boot-starter-web, spring-boot-starter-test. You can use spring-boot-starter-web to enable Spring MVC in Spring Boot application.

**Q: What are some common Spring Boot annotations?**

Some of the most common Spring Boot annotations are @EnableAutoConfiguration, @SpringBootApplication, @SpringBootConfiguration, and @SpringBootTest.

The **@EnableAutoConfiguration** is used to enable auto-configuration on Spring Boot application, while **@SpringBootApplication** is used on the Main class to allow it to run a JAR file. **@SpringBootTest** is used to run unit test on Spring Boot environment.

**Q: What is the difference between an embedded container and a WAR?**

The main difference between an embedded container and a WAR file is that you can Spring Boot application as a JAR from the command prompt without setting up a web server. But to run a WAR file, you need to first set up a web server like Tomcat which has Servlet container and then you need to deploy WAR there.

**Q: What is the difference between @SpringBootApplication and @EnableAutoConfiguration annotation? :** Even though both are essential Spring Boot application and used in the Main class or Bootstrap class there is a subtle difference between them. The @EnableAutoConfiguration is used to enable auto-configuration but @SpringBootApplication does a lot more than that.

It also combines **@Configuration and @ComponentScan** annotations to enable Java-based configuration and component scanning in your project.

The **@SpringBootApplication** is in fact combination of **@Configuration, @ComponentScan and @EnableAutoConfiguration** annotations. You can also check Spring Boot MasterClass to learn more about this annotation and it's used.

# **Spring Component Scan :** When developing Spring Boot applications, you need to tell the Spring Framework where to look for Spring components. Using component scan is one method of asking Spring to detect Spring-managed components. Spring needs the information to locate and register all the Spring components with the application context when the application starts.

Spring can auto scan, detect, and instantiate components from pre-defined project packages. Spring can auto scan all classes annotated with the stereotype annotations **@Component, @Controller, @Service, and @Repository**

Let's create a simple Spring Boot application to understand how component scanning works in Spring. We start by writing few components.

import org.springframework.stereotype.Component;

**@Component("demoBeanA")**

public class DemoBeanA {

}

Like wise we have components

**@Component("demoBeanB1")**

**@Component("demoBeanB2")**

**@Component("demoBeanC”)**

**@Component("demoBeanD")**

**public class DemoBeanD {**

**}**

## **The @SpringBootApplication Annotation :** Spring needs to know which packages to scan for annotated components in order to add them to the IoC container. In a Spring Boot project, we typically set the main application class with the @SpringBootApplication annotation. Under the hood, **@SpringBootApplication** is a composition of the @**Configuration, @ComponentScan, and @EnableAutoConfiguration**annotations. With this default setting, Spring Boot will auto scan for components in the current package (containing the @SpringBoot main class) and its sub packages.

The @ComponentScan annotation is used with the @Configuration annotation to tell Spring the packages to scan for annotated components. @ComponentScan also used to specify base packages and base package classes using the basePackageClasses or basePackages attributes of @ComponentScan.

The basePackageClasses attribute is a type-safe alternative to basePackages. When you specify basePackageClasses, Spring will scan the package (and subpackages) of the classes you specify.

**@Configuration**

**@ComponentScan(basePackages** = {

"guru.springframework.blog.componentscan.example.demopackageA",

"guru.springframework.blog.componentscan.example.demopackageD",

"guru.springframework.blog.componentscan.example.demopackageE"

},

basePackageClasses = DemoBeanB1.class)

public class BlogPostsApplicationWithComponentScan {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationWithComponentScan.class, args);

System.out.println("Contains A " + context.containsBeanDefinition("demoBeanA"));

System.out.println("Contains B2 " + context.containsBeanDefinition("demoBeanB2"));

System.out.println("Contains C " + context.containsBeanDefinition("demoBeanC"));

System.out.println("Contains D " + context.containsBeanDefinition("demoBeanD"));

}

}

The @ComponentScan annotation uses the basePackages attribute to specify three packages (and subpackages) that will be scanned by Spring. The annotation also uses the basePackageClasses attribute to declare the DemoBeanB1 class whose package Spring Boot should scan.

As demoBeanC is in a different package, Spring did not find it during component scanning.

## **Component Scanning Filters :** You can configure component scanning by using different type filters that Spring provides.

Filters can be of two types: **include and exclude filters**. As their names suggest, include filters specify which types are eligible for component scanning, while exclude filters specify which types are not.

You can use the include and/or exclude filters with or without the default filter. To disable the default filter, set the useDefaultFilters element of the @ComponentScan annotation to false.

@Configuration

@ComponentScan(value = "guru.springframework.blog.componentscan.example.demopackageA",

**useDefaultFilters** = false)

public class BlogPostsApplicationDisablingDefaultFilters {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationDisablingDefaultFilters.class,args);

System.out.println("Contains A " + context.containsBean("demoBeanA"));

}

}

In the preceding code, the value member defines the specific guru.springframework.blog.componentscan.example.demopackageA package to scan, while the useDefaultFilters member disables the default filter.

**Component Scanning Filter Types :** Spring provides the FilterType enumeration for the type filters that may be used in conjunction with @ComponentScan.

The available FilterType values are:

* **FilterType.ANNOTATION**: Include or exclude those classes with a stereotype annotation
* **FilterType.ASPECTJ**: Include or exclude classes using an AspectJ type pattern expression
* **FilterType.ASSIGNABLE\_TYPE:** Include or exclude classes that extend or implement this class or interface
* **FilterType.REGEX**: Include or exclude classes using a regular expression
* **FilterType.CUSTOM**: Include or exclude classes using a custom implementation of the org.springframework.core.type.TypeFilter interface

### **Include Filters :** With include filters, you can include certain classes to be scanned by Spring. To include assignable type, use the includeFilters element of the @ComponentScan annotation with FilterType.ASSIGNABLE\_TYPE. Using this filter, you can instruct Spring to scan for classes that extends or implements the class or interface you specify.

@Configuration

**@ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",**

**"guru.springframework.blog.componentscan.example.demopackageB"},**

**includeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, value = DemoBeanB2.class),**

**useDefaultFilters = false)**

public class BlogPostsApplicationIncludeFilter {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationIncludeFilter.class,args);

System.out.println("Contains A " + context.containsBean("demoBeanA"));

System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));

System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));

System.out.println("Contains B3 " + context.containsBean("demoBeanB3"));

}}

**Exclude Filters :** The @ComponentScan annotation enables you to exclude those classes that you do not want to scan.

@Configuration

@ComponentScan(basePackageClasses = {DemoBeanB1.class},

**excludeFilters** = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE,

value = DemoBeanB2.class))

In this code, the nested annotation @ComponentScan.Filter is used to specify the filter type as FilterType.ASSIGNABLE\_TYPE and the base class that should be excluded from scanning.

**Bean Overriding :** Spring beans are identified by their names within an ApplicationContext.

Thus, bean overriding is a default behavior that happens when we define a bean within an ApplicationContext which has the same name as another bean. It works by simply replacing the former bean in case of a name conflict.

Starting in Spring 5.1, the **[BeanDefinitionOverrideException](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/support/BeanDefinitionOverrideException.html)**was introduced to allow developers to automatically throw the exception to prevent any unexpected bean overriding. By default, the original behavior is still available which allows bean overriding.

Spring's @Bean annotation is a very common way of defining a bean.

Thus, another option is to set the name property of @Bean annotation:

**@Bean("testBean1")**

public TestBean1 testBean() {

return new TestBean1();

}

public TestBean1 testBean() {

return new TestBean2();

}

Another way to define a bean is with stereotype annotations. With Spring's @ComponentScan feature enabled, we can define our bean names at the class level using the @Component annotation:

**@Component("testBean1")**

class TestBean1 {

private String name; // standard getters and setters

}

**@Component("testBean2")**

class TestBean2 {

private String name; // standard getters and setters

}

**To enable bean overriding, let's set the** spring.main.allow-bean-definition-overriding property to true in our application.properties file:

**spring.main.allow-bean-definition-overriding=true**

By doing this, we are telling Spring Boot to allow bean overriding without any change to bean definitions.

**Spring Boot Under the Hood :** When we do a **SpringApplication.run(ApplicationBootClass.class)**, it starts the application.

How are Beans getting created (BeanFactory or ApplicationContext)? : When SpringApplication.run() command is invoked, the Application Context is created by calling createApplicationContext() method.

public ConfigurableApplicationContext **run**(String... args) {

// Create, load, refresh, and run the ApplicationContext

context = createApplicationContext();

return context ; // handle to the context object for the developer

}

**What exactly is the type of this context? :** The createApplicationContext method checks if it is a web or standalone application based on the type it creates for the context. I was creating a REST-based controller for which a context of type**AnnotationConfigEmbeddedWebApplicationContext** was initialized. In the case of a standalone application, **AnnotationConfigApplicationContext** will be initialized.

**How are the beans created once the context is initialized? :** When the constructor of the context is invoked, it will register the annotated class beans with the context. That's why no XML configurations are required. All your **@Repository, @Component, @Service, and Controller** beans will be registered and the context is returned.

The following lines of code are executed for context initialization and bean creation for a web application.

public **AnnotationConfigEmbeddedWebApplicationContext**(Class<?>... annotatedClasses) {

this();

register(annotatedClasses);

refresh(); // Refreshing org.springframework.boot.context.embedded. This log appears in the console}

**Which servlet acts as a front controller?** : **DispatcherServlet**.

The AnnotationConfigEmbeddedWebApplicationContext class extends the EmbeddedWebApplicationContext, which registers the dispatcher servlet.

public static final String DISPATCHER\_SERVLET\_NAME = ServletContextInitializerBeans.DISPATCHER\_SERVLET\_NAME;

**What about the embedded Tomcat? :** Normally, starting an embedded Tomcat is as easy as instantiating the Tomcat class.

So with regards to Spring Boot. the EmbeddedWebApplicationContext creates an instance of org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainer and adds the context.

TomcatEmbeddedServletContainer class has Tomcat as an instance variable.

Check the selfInitialize() method and prepareEmbeddedWebApplicationContext of the EmbeddedWebApplicationContext class:

prepareEmbeddedWebApplicationContext() {

servletContext.log("Initializing Spring embedded WebApplicationContext"); // these logs are printed in your STS console.

logger.info("Root WebApplicationContext: initialization completed in " );  // these logs are printed in your STS console.

}

**Configure a Spring Boot Web Application :The Port Number :** In main standalone applications, the main HTTP port defaults to 8080; we can easily configure Boot to use a different port:

server.port=8083

And for, YAML-based configuration:

server:

port: 8083

We can also programmatically customize the server port:

@Component

public class CustomizationBean implements

**WebServerFactoryCustomizer**<ConfigurableServletWebServerFactory> {

@Override

public void **customize**(ConfigurableServletWebServerFactory container) {

container.setPort(8083); }}

**The Context Path :** By default, Spring boot applications are accessed by context path “/” which is default for embedded servers i.e. we can access the application directly at http://localhost:PORT/.

But in production, we will deploy the application under some context root – so that we can refer the URLs for other places. Also, it is desirable to configure security and there we will need application’s context root.

By default, the context path is “/”. If that's not ideal and you need to change it – to something like /app\_name, here's the quick and simple way to do it via properties:

**server.servlet.contextPath**=/springbootapp

And for YAML-based configuration:

server: servlet: contextPath:/springbootapp

Finally – the change can be done programmatically as well: In Spring boot 2.x,

@Component

public class CustomizationBean implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

@Override

public void customize(ConfigurableServletWebServerFactorycontainer) {

container.setContextPath("/springbootapp");

}

}

application.properties

### Spring boot 1.x #########

server.contextPath=/ClientApp

### Spring boot 2.x #########

server.servlet.context-path=/ClientApp

**Shut Down a Boot Application Programmatically :** You can programmatically shut down a Boot app with the help of SpringApplication. This has a static exit() method that takes two arguments: the ApplicationContext and an ExitCodeGenerator:

@Autowired

public void shutDown(ExecutorServiceExitCodeGenerator exitCodeGenerator) {

SpringApplication.exit(applicationContext, exitCodeGenerator);

}

It's through this utility method that we can shut down the app.

**Configure the Logging Levels :** You can easily tune the logging levels in a Boot application; Starting with version 1.2.0 onwards, you can configure the log level in the main properties file:

logging.level.org.springframework.web: DEBUG

logging.level.org.hibernate: ERROR

**Configure Jetty or Undertow in Boot Application :** The Spring Boot starters generally use Tomcat as the default embedded server. If that needs to be changed – you can exclude the Tomcat dependency and include Jetty or Undertow instead:

**Configuring Jetty :**

<dependency>

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

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

**<exclusions>**

**<exclusion>**

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

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

**</exclusion>**

**</exclusions>**

</dependency>

<dependency>

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

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

</dependency>

**@Bean**

public JettyEmbeddedServletContainerFactory jettyEmbeddedServletContainerFactory() {

JettyEmbeddedServletContainerFactory jettyContainer =

new JettyEmbeddedServletContainerFactory();

jettyContainer.setPort(9000);

jettyContainer.setContextPath("/springbootapp");

return jettyContainer; }

**Spring Boot Actuator :** Spring boot’s module **Actuator** allows you to monitor and manage application usages in production environment, without coding and configuration for any of them. These monitoring and management information is exposed via REST like endpoint URLs.

**Actuator Maven Dependency :**

<dependency>

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

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

</dependency>

**Important Actuator Endpoints :** Most applications exposes endpoints via HTTP, where the ID of the endpoint along with a prefix of **/actuator** is mapped to a URL. For example, by default, the health endpoint is mapped to **/actuator/health**.

By default, only /health and /info are exposed via Web APIs. Rest are exposed via JMX. Use **management.endpoints.web.exposure.include=\*** to expose all endpoints through the Web APIs.

**application.properties :**

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

# To expose only selected endpoints

#management.endpoints.jmx.exposure.include=health,info,env,beans

**Some of important and widely used actuator endpoints are given below:-**

|  |  |
| --- | --- |
| **ENDPOINT** | **USAGE** |
| /auditevents | Returns all auto-configuration candidates and the reason why they ‘were’ or ‘were not’ applied. |
| /beans | Returns a complete list of all the Spring beans in your application. |
| /mappings | Displays a collated list of all @RequestMapping paths.. |
| /env | Returns list of properties in current environment |
| /health | Returns application health information. |
| /caches | It exposes available caches. |
| /conditions | Shows the conditions that were evaluated on configuration and auto-configuration. |
| /configprops | It displays a collated list of all @ConfigurationProperties. |
| /integrationgraph | It shows the Spring Integration graph. Requires a dependency on spring-integration-core. |
| /loggers | The configuration of loggers in the application. |
| /scheduledtasks | Displays the scheduled tasks in the application. |
| /sessions | Returns trace logs (by default the last 100 HTTP requests). Requires an HttpTraceRepository bean. |
| /httptrace | It allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a Servlet-based web application using Spring Session. |
| /shutdown | Lets the application be gracefully shutdown. Disabled by default. |
| /threaddump | It performs a thread dump. |
| /metrics | It shows several useful metrics information like JVM memory used, system CPU usage, open files, and much more. |
| /heapdump | Returns an hprof heap dump file. |
| /logfile | Returns the contents of the logfile if logging.file.name or logging.file.path properties have been set. |

**Securing Endpoints :** By default, spring security is enabled for all actuator endpoints if it available in the classpath.

If you wish to configure custom security for HTTP endpoints, for example, only allow users with a certain role to access then configure **WebSecurityConfigurerAdapter** in following manner:

@Configuration(proxyBeanMethods = false)

public class ActuatorSecurity extends **WebSecurityConfigurerAdapter** {

@Override

protected void **configure**(HttpSecurity http) throws Exception {

http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests) ->

requests.anyRequest().hasRole("ENDPOINT\_ADMIN"));

http.httpBasic();

}

}

The above configuration ensures that only users with role ENDPOINT\_ADMIN have access to actuation endpoints.

**Enabling Endpoints :** By default, all endpoints (except /shutdown) are enabled. To disable all endpoints, by default, use property:

Disable all endpoints by default :

**management.endpoints.enabled-by-default=false**

Then use the only required endpoints which the application need to expose using the pattern management.endpoint.<id>.enabled.

Enable only needed endpoints

**management.endpoint.health.enabled=true**

**management.endpoint.loggers.enabled=true**

**CORS support :** CORS support is disabled by default and is only enabled once the endpoints.cors.allowed-origins property has been set.

management.endpoints.web.cors.allowed-origins=https://example.com

management.endpoints.web.cors.allowed-methods=GET,POST

Here the management context path is /management.

**Caching the Response :** Actuator endpoints automatically cache the responses to read operations that do not take any parameters. Use cache.time-to-live property to configure the amount of time for which an endpoint will cache the response.

Caching the response for 20 seconds

management.endpoint.beans.cache.time-to-live=20s

**<http://localhost:8080/actuator/env> :** This will give all the environmental configuration about the server.

#### **<http://localhost:8080/actuator/beans> :** This will give all the spring beans loaded in the context.

**<http://localhost:8080/actuator/threaddump> :** This will give the current server thread dump.

**Change the Management endpoint context path :**By default all endpoints comes in default context path of the application, suffixed with /actuator. If for some reason, we have existing endpoints in application starting with /actuator then we can customize the base path to something else.

All we need to specify the new base path in the application.properties.

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

Now you will be able to access all actuator endpoints under a new URL. e.g.

/manage/health

/manage/dump

/manage/env

/manage/beans

**Customize the management server port :**To customize the management endpoint port, we need to add this entry in the application.properties file.

management.server.port=8081

## **Spring Boot – Get all loaded beans with Class Type Information :** Spring boot loads lots of beans internally to run your application with minimal configuration.

To execute a method automatically, when application is fully loaded, I am using CommandLineRunner interface. [CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html) is used to indicate that a bean should run when it is contained within a Spring Application.

1) Use **ApplicationContext.getBeanDefinitionNames()**to find the name of all loaded beans  
2) Use **ApplicationContext.getBean(beanName)** to get bean including its runtime type information.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.builder.SpringApplicationBuilder;

import org.springframework.boot.web.support.SpringBootServletInitializer;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class SpringBootWebApplication extends SpringBootServletInitializer implements **CommandLineRunner** {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(SpringBootWebApplication.class);

}

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

SpringApplication.run(SpringBootWebApplication.class, args);

}

@Autowired

private ApplicationContext appContext;

@Override

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

{

String[] beans = appContext.**getBeanDefinitionNames**();

Arrays.sort(beans);

for (String bean : beans)

{

System.out.println(bean + " of Type :: " + appContext.getBean(bean).getClass());

}

}

}

**Spring Boot - Exception Handling :** Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications.

**Controller Advice :** The **@ControllerAdvice** is an annotation, to handle the exceptions globally.

**Exception Handler :** The **@ExceptionHandler** is an annotation used to handle the specific exceptions and sending the custom responses to the client.

**Step 1 :** Create @ControllerAdvice class to handle the exceptions globally −

@ControllerAdvice

public class ProductExceptionController {

}

**Step 2 :** Define a Exception class that extends the RuntimeException class.

public class ProductNotfoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

}

**Step 3 :** You can define the @ExceptionHandler method to handle the exceptions as shown. This method should be used for writing the Controller Advice class file.

@ExceptionHandler(value = ProductNotfoundException.class)

public ResponseEntity<Object> **exception**(ProductNotfoundException exception) {

}

**Step 4 :** Now, use the code given below to throw the exception from the API.

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object> updateProduct() {

throw new ProductNotfoundException();

}

**Full code :**

@ControllerAdvice

public class ProductExceptionController {

@ExceptionHandler(value = ProductNotfoundException.class)

public ResponseEntity<Object> exception(ProductNotfoundException exception) {

return new ResponseEntity<>("Product not found", HttpStatus.NOT\_FOUND);

}

}

**Controller Class :**

@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)

public ResponseEntity<Object> updateProduct(@PathVariable("id") String id, @RequestBody Product product) {

if(!productRepo.containsKey(id))throw new ProductNotfoundException();

productRepo.remove(id);

product.setId(id);

productRepo.put(id, product);

return new ResponseEntity<>("Product is updated successfully", HttpStatus.OK);

}

**Example :**

 I have created **RecordNotFoundException**class for all such scenarios where a resource is requested by it’s ID, and resource is not found in the system.

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class **RecordNotFoundException** extends RuntimeException {

public RecordNotFoundException(String exception) {

super(exception);

}

}

**Custom ExceptionHandler :** Now add one class extending **ResponseEntityExceptionHandler**and annotate it with **@ControllerAdvice** annotation.

**ResponseEntityExceptionHandler**is a convenient base class for to provide centralized exception handling across all @RequestMapping methods through @ExceptionHandler methods. @ControllerAdvice is more for enabling auto-scanning and configuration at application startup.

@ControllerAdvice

public class **CustomExceptionHandler** extends **ResponseEntityExceptionHandler**

{

@**ExceptionHandler**(Exception.class)

public final ResponseEntity<Object> **handleAllExceptions**(Exception ex, WebRequest request) {

List<String> details = new ArrayList<>();

details.add(ex.getLocalizedMessage());

ErrorResponse error = new ErrorResponse("Server Error", details);

return new ResponseEntity(error, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

@**ExceptionHandler**(RecordNotFoundException.class)

public final ResponseEntity<Object> handleUserNotFoundException(RecordNotFoundException ex, WebRequest request) {

List<String> details = new ArrayList<>();

details.add(ex.getLocalizedMessage());

ErrorResponse error = new ErrorResponse("Record Not Found", details);

return new ResponseEntity(error, HttpStatus.NOT\_FOUND);

}

@Override

protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

List<String> details = new ArrayList<>();

for(ObjectError error : ex.getBindingResult().getAllErrors()) {

details.add(error.getDefaultMessage());

}

ErrorResponse error = new ErrorResponse("Validation Failed", details);

return new **ResponseEntity**(error, HttpStatus.BAD\_REQUEST);

}

}

Above class handles multiple exceptions including RecordNotFoundException; and it also handle request validation errors in @RequestBody annotated object.

**Spring boot exception handling – REST request validation**

**Default spring validation support :** To apply default validation, we only need to add relevant annotations in proper places. i.e.

* Annotate model class with required validation specific annotations such as @NotEmpty, @Email etc.

@NotEmpty(message = "first name must not be empty")

private String firstName;

@NotEmpty(message = "last name must not be empty")

private String lastName;

@NotEmpty(message = "email must not be empty")

@Email(message = "email should be a valid email")

private String email;

|  |  |
| --- | --- |
| **ANNOTATION** | **USAGE** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**You should return a proper error response :**

Clear message indicating what went wrong and what the consumer can do to fix the error.

Include information necessary to solve the error.

Proper Response Status based on the context.

Do not include sensitive information in the response.

**Response Statuses for Errors :**

Use appropriate status code based on the error.

404 - RESOURCE NOT FOUND

400 - BAD REQUEST

401 - UNAUTHORIZED

415 - UNSUPPORTED TYPE - Representation not supported for the resource

500 - SERVER ERROR

**Let’s consider a few HTTP Methods:**

**GET** : Should not update anything. Should be idempotent (same result in multiple calls). Possible Return Codes 200 (OK) + 404 (NOT FOUND) +400 (BAD REQUEST)

**POST** : Should create new resource. Ideally return JSON with link to newly created resource. Same return codes as get possible. In addition - Return code 201 (CREATED) can be used.

**PUT** : Update a known resource. ex: update client details. Possible Return Codes : 200(OK) + 404 (NOT FOUND) +400 (BAD REQUEST)

**DELETE** : Used to delete a resource. Possible Return Codes : 200(OK).

By default, Spring boot applications start with embedded tomcat server start at default port 8080. We can change default embedded server port to any other port, using any one of below technique.

TIP – To scan for a free port (using OS natives to prevent clashes) use server.port=0. Now spring boot will find any unassigned http port for us.

1. Change default **server port** from properties file

We can do lots of wonderful things by simply making few entries in application properties file in any spring boot application. Changing server port is one of them.

application.properties

application.properties

server.port=9000

**Change the server port programatically**

EmbeddedServletContainerCustomizer interface is used to customize embedded tomcat configuration. Any beans of this type will get a callback with the container factory before the container itself is started, so we can set the port, address, error pages etc.

**Spring boot2 – WebServerFactoryCustomizer interface**

Change default server port in spring boot2 applications by implementing ConfigurableWebServerFactory interface.

@Component

public class AppContainerCustomizer

implements WebServerFactoryCustomizer< ConfigurableWebServerFactory > {

@Override

public void customize(ConfigurableWebServerFactory factory) {

factory.setPort(9000);

}

}

**Spring boot 1.x – EmbeddedServletContainerCustomizer interface**

Change default server port in spring boot 1.x applications by implementing EmbeddedServletContainerCustomizer interface.

@Component

public class AppContainerCustomizer implements EmbeddedServletContainerCustomizer {

@Override

public void customize(ConfigurableEmbeddedServletContainer container)

container.setPort(9000);

}

}

# **Spring Boot – Configure Jetty Server :** Add spring-boot-starter-jetty dependency : You will need to update pom.xml and add dependency for spring-boot-starter-jetty. Also, you will need to exclude default added spring-boot-starter-tomcat dependency.

**Configure Jetty Options :**

To override, default jetty runtime configuration – you can configure them in application.properties file.

**application.properties**

server.port=8080

server.servlet.context-path=/home

####Jetty specific properties########

server.jetty.acceptors= # Number of acceptor threads to use.

server.jetty.max-http-post-size=0 # Maximum size in bytes of the HTTP post or put content.

server.jetty.selectors= # Number of selector threads to use.

Also, you may configure these options programatically using JettyEmbeddedServletContainerFactory bean.

Bean

public JettyEmbeddedServletContainerFactory  jettyEmbeddedServletContainerFactory() {

    JettyEmbeddedServletContainerFactory jettyContainer =

        new JettyEmbeddedServletContainerFactory();

    jettyContainer.setPort(9000);

jettyContainer.setContextPath("/home");     return jettyContainer;}

**Spring Boot SSL [https] Terminology:**

****SSL**** – stands for Secure Sockets Layer. It is the industry standard protocol for keeping an internet connection secure by safeguarding all sensitive data that is being sent between two systems, preventing hackers from reading and modifying any information transferred.

****TLS**** – (Transport Layer Security) is an updated, more secure, version of SSL. It adds more features. Today, certificates provided by certificate authorities are based on TLS only. But regarding secured communication over network, the term SSL is still common as it is the old and just become popular among community.

****HTTPS**** – (Hyper Text Transfer Protocol Secure) appears in the URL when a website is secured by an SSL certificate. It is the secured version of HTTP protocol.

****Truststore and Keystore**** – Those are used to store SSL certificates in Java but there is little difference between them. truststore is used to store public certificates while keystore is used to store private certificates of client or server.

Spring boot HTTPS Config :

server.port=8443

server.ssl.key-alias=selfsigned\_localhost\_sslserver

server.ssl.key-password=changeit

server.ssl.key-store=classpath:ssl-server.jks

server.ssl.key-store-provider=SUN

server.ssl.key-store-type=JKS

**Create your own self signed SSL certificate :**To get SSL digital certificate for our application we have two options –

to create a self-signed certificate

to obtain SSL certificate from certification authority(CA) we call it CA certificate.

For today’s demo purpose we will create self-signed certificate generated by java keytool command. We need to run the keytool -genkey command from command prompt.

Here is the exact command we will use –

keytool -genkey -alias selfsigned\_localhost\_sslserver -keyalg RSA -keysize 2048 -validity 700 -keypass changeit -storepass changeit -keystore ssl-server.jks

**Let’s understand above command –**

-genkey – is the keytool command to generate the certificate, actually keytool is a multipurpose and robust tool which has several options

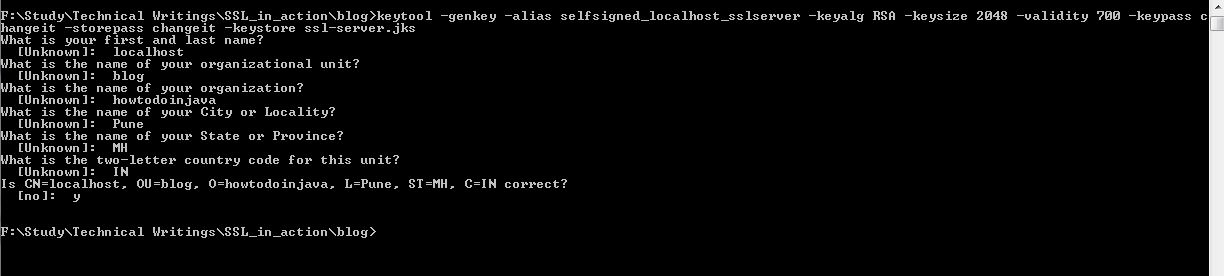
-alias selfsigned\_localhost\_sslserver – indicates the alias of the certificate, which is used by SSL/TLS layer

-keyalg RSA -keysize 2048 -validity 700 – are self descriptive parameters indicating the crypto algorithm, keysize and certificate validity.

-keypass changeit -storepass changeit – are the passwords of our truststore and keystore

-keystore ssl-server.jks – is the actual keystore where the certificate and public/private key will be stored. Here we are using JKS fromat – Java Key Store, there are other formats as well for keystore.

Once we execute above command, it will ask for certain information and finally this will look like this.



That’s all we need at this point regarding certification generation. This will generate the ssl-server.jks keystore file containing our self signed certificates in the directory from where keytool command has been executed.

To view what is inside this keystore we can again use the keytool -list command as bellow.

keytool -list -keystore ssl-server.jks

**Spring boot SSL Configuration**

First we need to copy the generated keystore file (ssl-server.jks) into the resources folder and then open the application.properties and add the below entries.

server.port=8443

server.ssl.key-alias=selfsigned\_localhost\_sslserver

server.ssl.key-password=changeit

server.ssl.key-store=classpath:ssl-server.jks

server.ssl.key-store-provider=SUN

server.ssl.key-store-type=JKS

For testing purpose we will use one simple REST endpoint. To do that open the already generated spring boot application class annotated with @SpringBootApplication and add this code. This will expose one rest endpoint with relative URL /secured in the server.

@RestController

class SecuredServerController{

@RequestMapping("/secured")

public String secured(){

System.out.println("Inside secured()");

return "Hello user !!! : " + new Date();

}

}

**Redirect HTTP requests to HTTPS :** This is an optional step in case you want to redirect your HTTP traffic to HTTPS, so that the full site becomes secured. To do that in spring boot, we need to add HTTP connector at 8080 port and then we need to set redirect port 8443. So that any request in 8080 through http, it would be automatically redirected to 8443 and https.

To do that you just need to add below configuration.

@Bean

public EmbeddedServletContainerFactory servletContainer() {

TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerFactory() {

@Override

protected void postProcessContext(Context context) {

SecurityConstraint securityConstraint = new SecurityConstraint();

securityConstraint.setUserConstraint("CONFIDENTIAL");

SecurityCollection collection = new SecurityCollection();

collection.addPattern("/\*");

securityConstraint.addCollection(collection);

context.addConstraint(securityConstraint);

}

};

tomcat.addAdditionalTomcatConnectors(redirectConnector());

return tomcat;

}

private Connector redirectConnector() {

Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");

connector.setScheme("http");

connector.setPort(8080);

connector.setSecure(false);

connector.setRedirectPort(8443);

return connector;

}

**Spring boot @Scheduled annotation example :** To schedule job in spring boot application to run periodically, spring boot provides @EnableScheduling and @Scheduled annotations. Lets learn to use Spring boot @Scheduled annotation.

Add @EnableScheduling to Spring Boot Application class : Add @EnableScheduling annotation to your spring boot application class. @EnableScheduling is a Spring Context module annotation. It internally imports the SchedulingConfiguration via the

@Import(SchedulingConfiguration.class) instruction

@SpringBootApplication

@EnableScheduling

public class SpringBootWebApplication {

}

Add Spring boot @Scheduled annotations to methods

Now you can add @Scheduled annotations on methods which you want to schedule. Only condition is that methods should be without arguments.

ScheduledAnnotationBeanPostProcessor that will be created by the imported SchedulingConfiguration scans all declared beans for the presence of the @Scheduled annotations.

For every annotated method without arguments, the appropriate executor thread pool will be created. This thread pool will manage the scheduled invocation of the annotated method.

**Schedule task at fixed rate :**Execute a task at a fixed interval of time:

@Scheduled(initialDelay = 1000, fixedRate = 10000)

public void run() {

logger.info("Current time is :: " + Calendar.getInstance().getTime());

}

**Spring boot cron job example :** @Scheduled annotation is very flexible and may take cron expression as well.

**@Scheduled(cron = "0 10 10 10 \* ?")**

public void run() {

logger.info("Current time is :: " + Calendar.getInstance().getTime());

}

**Spring @ConditionalOnProperty Example :** The @ConditionalOnProperty annotation allows you to load beans conditionally depending on a certain environment property or configuration of a property.

**@ConditionalOnProperty** annotation is used to check if specified property available in the environment or it matches some specific value so it can control the execution of some part of code like bean creation. It may be useful in many cases for example enable/disable service if specific property is available. Below are the attributes which can be used for property check.

* **havingValue -** Provide the value which need to check against specified property otherwise it will check that value should not be false.
* **matchIfMissing** - If true it will match the condition and execute the annotated code when property itself is not available in environment.
* **name** - Name of the property to be tested. If you want to test single property then you can directly put the property name as string like "property.name" and if you have multiple properties to test then you can put the names like {"prop.name1","prop.name2"}
* **prefix** - It can be use when you want to apply some prefix to all properties. For example in case of above properties we can mention like (prefix="prop", name={"name1","name2"}.

**Example:**

@Configuration

public class SpringConfig {

@Bean

@ConditionalOnProperty(prefix = "module", name = "enabled", matchIfMissing = true)

public SpringService springService() {

return new SpringService();

}

}

I have used here prefix and name to denote the configuration property. Therefore the actual property name is module.enabled.

I have matchIfMissing = true in the @ConditionalOnProperty and it means that if the property module.enabled does not exist, it will still be loaded.

The SpringService is only loaded if the module.enabled property has the value true. So in the following scenario the SpringService will be loaded.

**Other Scenarios**

For example let’s remove the tag matchIfMissing = true, i.e., **@ConditionalOnProperty(prefix = "module", name = "enabled")** and execute the main class. You will get the following error:

The following candidates were found but could not be injected:

- Bean method 'springService' in 'SpringConfig' not loaded because @ConditionalOnProperty (module.enabled) did not find property 'enabled'

**Why**? Because you removed the matchIfMissing = true and you did not configure the property in properties file.

So put the property **module.enabled=true** into **src/main/resources/application.properties file**.

Now executing main class will not give any error and SpringService bean will be loaded.

Let’s say you have configured with havingValue = "true" as shown below:

@ConditionalOnProperty(prefix = "module", name = "enabled", havingValue = "true")

So if the property has value true then only SpringService bean will be loaded.

Now if you make havingValue = "false" then your bean SpringService will not be loaded and you will see below error:

The following candidates were found but could not be injected:

- Bean method 'springService' in 'SpringConfig' not loaded because @ConditionalOnProperty (module.enabled=false) found different value in property 'enabled'

If you do not configure the property in properties file and havingValue = "true" with matchIfMissing = true does not make sense.

You can also use only @ConditionalOnProperty(prefix = "module", value = "enabled") to load the SpringService because the value of module.enabled is true (module.enabled=true) in the properties file.

But if you configure as module.enabled=false, then SpringService bean will not be loaded and you will get below error.

The following candidates were found but could not be injected:

- Bean method 'springService' in 'SpringConfig' not loaded because @ConditionalOnProperty (module.enabled) found different value in property 'enabled'

You can also pass as value in place of name. You can also pass multiple names or multiple values but prefix will be only one.

To pass value instead of name, you can use @ConditionalOnProperty(prefix = "module", value = "enabled").

To pass multiple names or values you can use @ConditionalOnProperty(prefix = "module", name = {"enabled", "dynamic"}).

**Multiple Databases in Spring Boot :**  By default, Spring Boot will instantiate its default DataSource with the configuration properties prefixed by spring.datasource.\*:

spring.datasource.jdbcUrl = [url]

spring.datasource.username = [username]

spring.datasource.password = [password]

We now want to keep on using the same way to configure the second DataSource, but with a different property namespace:

spring.second-datasource.jdbcUrl = [url]

spring.second-datasource.username = [username]

spring.second-datasource.password = [password]

Because we want the Spring Boot autoconfiguration to pick up those different properties (and instantiate two different DataSources), we'll define two configuration classes.

@Configuration

@PropertySource({"classpath:persistence-multiple-db-boot.properties"})

@EnableJpaRepositories(

basePackages = "com.baeldung.multipledb.dao.user",

entityManagerFactoryRef = "userEntityManager",

transactionManagerRef = "userTransactionManager")

public class PersistenceUserAutoConfiguration {

@Primary

@Bean

@ConfigurationProperties(prefix="spring.datasource")

public DataSource userDataSource() {

return DataSourceBuilder.create().build();

}

// userEntityManager bean

// userTransactionManager bean

}

@Configuration

@PropertySource({"classpath:persistence-multiple-db-boot.properties"})

@EnableJpaRepositories(

basePackages = "com.baeldung.multipledb.dao.product",

entityManagerFactoryRef = "productEntityManager",

transactionManagerRef = "productTransactionManager")

public class PersistenceProductAutoConfiguration {

@Bean

**@ConfigurationProperties(prefix="spring.second-datasource")**

public DataSource productDataSource() {

return DataSourceBuilder.create().build();

}

// productEntityManager bean

// productTransactionManager bean

}

We have defined the data source properties inside p**ersistence-multiple-db-boot.properties** according to the Boot auto-configuration convention.

The interesting part is annotating the data source bean creation method with **@ConfigurationProperties**. We just need to specify the corresponding config prefix. Inside this method, we're using a DataSourceBuilder, and Spring Boot will automatically take care of the rest.

But how do the configured properties get injected into the DataSource configuration?

When calling the build() method on the DataSourceBuilder, it'll call its private bind() method:

public T build() {

Class<? extends DataSource> type = getType();

DataSource result = BeanUtils.instantiateClass(type);

maybeGetDriverClassName();

bind(result);

return (T) result;

}

This private method performs much of the autoconfiguration magic, binding the resolved configuration to the actual DataSource instance:

private void bind(DataSource result) {

ConfigurationPropertySource source = new MapConfigurationPropertySource(this.properties);

ConfigurationPropertyNameAliases aliases = new ConfigurationPropertyNameAliases();

aliases.addAliases("url", "jdbc-url");

aliases.addAliases("username", "user");

Binder binder = new Binder(source.withAliases(aliases));

binder.bind(ConfigurationPropertyName.EMPTY, Bindable.ofInstance(result));

}