**Spring Boot** is basically an extension of the Spring framework, which provides rapid application development, where we can develop stand-alone web-based applications.

**Features of spring Boot**

Autoconfiguration

EmbededServer

Actuators

Starters

Spring-Profiles

dev-tools

**Difference between Spring and Springboot?**

**Spring:**

1.Spring is a framework to develop web-based applications with java as a language.

2.If we want to run the application, we need to add the server explicitly.

3.We need to do the configurations manually.

4.We require XML file to do the configurations.

5.We need to add the each and every dependency manually.

6.It takes time to develop any application when working with Spring.

Spring Boot streamlines the development of Spring applications by providing pre-configured templates, automatic configurations, embedded servers, and built-in tools for monitoring and management. It reduces the amount of boilerplate code and configuration required, allowing developers to focus more on building business logic. Traditional Spring, while powerful and flexible, requires more manual configuration and setup.

**SpringBoot (features):**

1.SpringBoot is an extended feature of Spring framework which provides rapid application development, where we can develop stand-alone web-based applications.

2.We can run the application directly without adding any server, because it contains embedded server like tomcat, jetty,Undertow etc.,

3.We don't need any XML file for configuration.

4.It provides a feature of Auto configuration.

5.It provides production ready features like bean info, health condition when you add actuator dependency.

6.It provides concept of starters where dependencies are added no need to add manually.

7.It reduces development time and increases production time when you develop a web-based application.

8.When we add devtool dependency in pom.xml, application is loaded automatically when we do any changes.

**Spring Boot autoconfiguration** represents a way to automatically configure a Spring application based on the dependencies that are present on the classpath. This can make development faster and easier by eliminating the need for defining certain beans that are included in the auto-configuration classes.

**AutoConfiguration and how it works internally?**

It is a feature of Springboot where it will automatically configure the beans with the application context based on the application related properties such as presence of certain classes in the class path, existence of beans and activation of some properties.

**Example:**

When we add SpringbootStarter Spring data Jpa, then in the auto configuration class based on the

@ConditionalOnClass, it will check whether DataSource class is available in class path or not.

@ConditionalOnBean, it will check whether bean with same name created or not.

@ConditinalOnProperty it will check whether the database properties are there in the application.properties .

Like based on these conditions' configuration is done and is registered with the application context.

Data source bean will be created automatically if we have SpringBoot -starter jpa on class path and database relation config info in properties file

spring.datasource.url=jdbc:mysql://localhost:3306/my\_java

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

RestTemplate will be crated automatically if we add Spring starter web dependency

ViewResolver Bean will cerated based on Spring starter web depency in class path and application prop file

spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp

**Disabling specific auto-configuration**

If you find that specific auto-configure classes are being applied that you don’t want, you can use the exclude attribute of @EnableAutoConfiguration to disable them.

*@Configuration*

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

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

}

If the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead. Finally, you can also control the list of auto-configuration classes to exclude via the spring.autoconfigure.exclude property.

### **What Is An Embedded Server?**

An embedded server is embedded as part of the deployable application.

If we talk about Java applications, that would be a JAR.

The advantage with this is you don’t need the server pre-installed in the deployment environment.

With SpringBoot, the default embedded server is **Tomcat**. Other options available are **Jetty** and **UnderTow.**

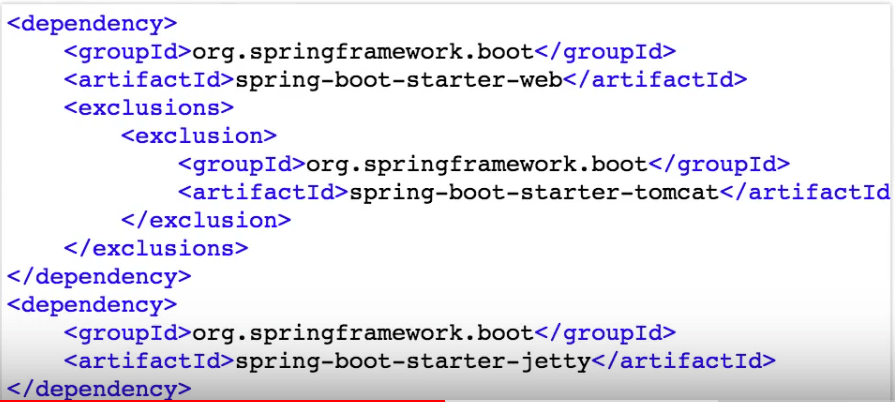
Embedded servers are quite scalable, and can host applications that support millions of users. These are no less scalable than the conventional fat servers. In a typical microservice architecture, there could be hundreds of microservice instances deployed at a given point in time.

We would like to automate development and deployment of microservices to the maximum extent possible.

A good approach would be to take the application, wrap it in a container image, and manage it as needed using something like Kubernetes.

### Switching To Jetty

By default, the Spring Boot framework uses Tomcat as the embedded server of choice. However, you could override this default setting by specifying certain configuration settings. For instance, if you want to use a Jetty dependency instead, then use an <exclusion> element in the XML configuration file, and specify a <dependency> element as well:



A similar dependency also exists for Undertow.

### What is Spring Actuator? What are its advantages?

An actuator is an additional feature of Spring that helps you to monitor and manage your application when you push it to production. These actuators include auditing, health, CPU usage, HTTP hits, and metric gathering, and many more that are automatically applied to your application.

### How to enable Actuator in Spring boot application?

To enable the spring actuator feature, we need to add the dependency of “spring-boot-starter-actuator” in pom.xml.

<dependency>

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

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

</dependency>

management.endpoints.web.exposure.include=\* // to enable all end points add this in prop file

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

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

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

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

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

### How to check the environment properties in your Spring boot application?

Spring Boot actuator “/env” returns the list of all the environment properties of running the spring boot application.

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

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

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

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

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

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

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

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

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

**Starters**

SpringBoot Starters are maven template, which contains collection of relevant dependencies that are needed to start any functionality, like we add springbootstarter spring data Jpa to do database operations, springbootstarter web to develop web-based applications etc.,

Dependency management is a critical aspects of any complex project. And doing this manually is less than ideal; the more time you spent on it the less time you have on the other important aspects of the project.

Spring Boot starters were built to address exactly this problem. Starter POMs are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy-paste loads of dependency descriptors.

We have more than 30 Boot starters available – let's see some of them in the following sections.

## Spring Boot Starter Parent

The spring-boot-starter-parent  is a special starter–

**It also provides default configuration for Maven plugins** (such as *maven-failsafe-plugin*, *maven-jar-plugin*, *maven-surefire-plugin*, *maven-war-plugin*.)

Beyond that, it also inherits dependency management from *spring-boot-dependencies*which is the parent to the s*pring-boot-starter-parent*.

Once, we declare the starter parent in our project**, we don't need to define versions of the dependencies, Maven will download jar files based on the version defined for starter parent in the parent tag.**

For example, if we're building a web project, we can add spring-boot-starter-web directly, and we don't need to specify the version:

## ****The Web Starter****

First, let's look at developing the REST service; we can use libraries like Spring MVC, Tomcat and Jackson – a lot of dependencies for a single application.

Spring Boot starters can help to reduce the number of manually added dependencies just by adding one dependency. So instead of manually specifying the dependencies just add one starter as in the following example:

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

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

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

Spring Boot Profiles

While developing the application we deal with multiple environments such as dev, QA, Prod, and each environment requires a different configuration. For eg., we might be using an embedded H2 database for dev but for prod, we might have proprietary Oracle or DB2. Even if DBMS is the same across the environment, the URLs will be different.

To make this easy and clean, Spring has the provision of Profiles to keep the separate configuration of environments.

**Need for Profiles**

Enterprise application development is complex. You have multiple environments

* Dev
* QA
* Stage
* Production

You want to have different application configuration in each of the environments.

*Profiles help to have different application configuration for different environments.*

Spring and Spring Boot provide features where you can specify

* What is the configuration for various environments in different profiles?
* Set the active profile for a specific environment.

Spring Boot would pick up the application configuration based on the active profile that is set in a specific environment.

* application.properties - Configuration for application. Active profile is set in application.properties in this example
* application-dev.properties - Configuration Overrides for dev profile
* application-prod.properties - Configuration Overrides for prod profile

**Setting Active Profile**

Once you have profile specific configuration, you would need to set the active profile in an environment.

There are multiple ways of doing this

* Using -Dspring.profiles.active=prod in VM Arguments
* Use spring.profiles.active=prod in application.properties

In this example let’s set it in application.properties. Lets add another property to application.properties

spring.profiles.active=dev

When you restart the application, you would see that the dev profile is active.

## Configuring Profile Specific Beans

You can take this one step further and configure profile specific beans that are created only in specific profiles.

Let’s add this to SpringBootTutorialBasicsConfigurationApplication.java

@Profile("dev")

@Bean

public String devBean() {

return "dev";

}

@Profile("qa")

@Bean

public String qaBean() {

return "qa";

}

@Profile("prod")

@Bean

public String prodBean() {

return "prod";

}

Using @Profile annotation we can indicate the active profile in which a specific bean should be created.

To test this let’s further enhance SpringBootTutorialBasicsConfigurationApplication.

Let’s print the name of all the beans that are loaded.

public static void main(String[] args) {

ApplicationContext applicationContext = SpringApplication

.run(SpringBootTutorialBasicsConfigurationApplication.class, args);

for (String name : applicationContext.getBeanDefinitionNames()) {

System.out.println(name);

}

}

We are currently using dev profile

spring.profiles.active=dev

When you reload the application, you would see the following in the log

devBean

Spring Boot DevTools

Spring Boot 1.3 provides another module called Spring Boot DevTools. DevTools stands for **Developer Tool.** The aim of the module is to try and improve the development time while working with the Spring Boot application. Spring Boot DevTools pick up the changes and restart the application.

We can implement the DevTools in our project by adding the following dependency in the pom.xml file.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-devtools**</artifactId>**
4. **<scope>**runtime**<scope** **>**
5. **</dependency>**

Spring Boot Application Flow

Flow of Spring Boot Application Starts from Our Main Class, where we will be having main method, we will be using @SpringBootApplication annotaion on top of the main class @SpringBoot application is combination of @configuration @EnableAutoConfiguration and @ComponentScan . In our Main class will be having a Main method form main method we will be calling SpringApplication.run method we will be passing our main class name and args as arguments SpringApplication.run(MyApplication.class, args); from run it will internally check the application Context type based on dependency’s in our class path and if it is a web application then it will create webapplicationannotationconfig web application context , after creating application context as we have used component scan it will scan for all the base packages for the classes annotated with @component @contoller @service @repository @bean on methods and then application context will create beans for this classes and register it , when it finds @autowired annotation it will inject the beans, as we have used @enableautofiguration it will check for the autoconfiguration classes based on some conditions like @condiationlon class @conditionalonbean @condiationalonpropery automatically beans will be crated like datasource, resttemplate etc… now at last internally will call tomcat and application will get started

=====

There is main class autogenerated and somehow it do some magic when we run the spring boot application, how come the main method initiate servlet, run the container and all that actually happened when spring boot called this main method. We will unfold this in below article about the **SpringApplication.run** method So lets set go!!.

When we run this class as a java application then our application gets started.

Let’s try to **unfold the magic** which goes behind this SpringApplication.run method.

SpringApplication.run(StudentManagementApplication.class, args), is a static method it **returns** an object of **ConfigurableApplicationContext.**

ConfigurableApplicationContext ctx=SpringApplication.run(StudentManagementApplication.class, args);

Thus spring container gets started once **SpringApplication.run()** method is called.

Also learn: [What is Spring boot Application?](http://learnspringboot.in/getting-start-with-spring-boot/)

Spring container once gets started is responsible for:

1. **Creating all objects:** This is done by component scan remember @SpringBootApplication is a combination of @Configuration + @ComponentScan + @EnableAutoConfiguration.
2. **Dependency Injection**.
3. Managing **the life cycle** of all beans.

Steps Executed under this method –

So in short when the main method runs following **steps** occur:

1. **Application Context** is started.
2. Using application context **autodiscovery** occurs: @ComponentScan
3. All default configurations are set up ie based on dependencies mentioned spring boot automatically sets up defaults. It makes use of intelligence that if we have included spring-web starter then dispatcher servlet is auto-configured. (**@EnableAutoConfiguration**)
4. An embedded servlet container is started. ( No need to set up a separate web server ) . Note embedded servlet container is launched only if the web is mentioned in a dependency

Also Read: [@SpringBootApplication Annotation](http://learnspringboot.in/understand-springbootapplication-annotation/)

So, next time you run the main method, I hope you remember the magic that goes behind is application context, that’s why we recommend you to learn **Spring Core as a prerequisite** to Spring Boot.

Please leave us a comment if you have any issue in understanding above.

[**200 OK**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200)

The request has succeeded. The meaning of the success depends on the HTTP method:

* GET: The resource has been fetched and is transmitted in the message body.
* HEAD: The representation headers are included in the response without any message body.
* PUT or POST: The resource describing the result of the action is transmitted in the message body.
* TRACE: The message body contains the request message as received by the server.

[**201 Created**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201)

The request has succeeded and a new resource has been created as a result. This is typically the response sent after POST requests, or some PUT requests.

[**400 Bad Request**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400)

The server could not understand the request due to invalid syntax.

[**401 Unauthorized**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401)

Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.

[**402 Payment Required**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/402)

This response code is reserved for future use. The initial aim for creating this code was using it for digital payment systems, however this status code is used very rarely and no standard convention exists.

[**403 Forbidden**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403)

The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401, the client's identity is known to the server.

[**404 Not Found**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404)

The server can not find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 to hide the existence of a resource from an unauthorized client. This response code is probably the most famous one due to its frequent occurrence on the web.

[**405 Method Not Allowed**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/405)

The request method is known by the server but is not supported by the target resource. For example, an API may forbid DELETE-ing a resource.

[**500 Internal Server Error**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500)

The server has encountered a situation it doesn't know how to handle.

[**502 Bad Gateway**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502)

This error response means that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.

[**503 Service Unavailable**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503)

The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent. This response should be used for temporary conditions and the Retry-After: HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary condition responses should usually not be cached.

[**504 Gateway Timeout**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/504)

This error response is given when the server is acting as a gateway and cannot get a response in time.