

**What is Spring Boot?**

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

**Features**

* Create stand-alone Spring applications.
* Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files).
* Provide opinionated 'starter' POMs to simplify your Maven configuration.
* Automatically configure Spring whenever possible.
* Provide production-ready features such as metrics, health checks and externalized configuration.
* Absolutely no code generation and no requirement for XML configuration.

**To make application as Spring Boot application:**

1. **Spring Boot started parent:** provides version to spring jar’s you can override using properties tag.

* Dependency Versioning (to avoid dependency version conflicts.)
* Default Plugins
* Java Version

<parent>

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

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

<version>1.4.0.RELEASE</version>

</parent>

<properties>

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

</properties>

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

1. **Spring Boot started web:** is used get lots of jar instead of adding one by one in maven. Tomcat is default embedded container.
2. **Spring Boot maven plugin:** is used to create war file and launch the application.
3. **Application Main Class:**

@SpringBootApplication

**public** **class** Application {

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

SpringApplication.*run*(Application.**class**, args);

}

}

**Spring Boot auto configure:** if jar’s is there in class path spring boot it will automatically register Dispatcher servlet, error page and other stuff.

To see all auto configuration add this in application.properties file.

logging.level.org.springframework=DEBUG

**Spring Boot starter web service:** is used to for SOAP web services.

**Spring Boot starter data rest:**

**Spring Boot vs Spring:**

**Applications with Spring Framework**

* Over the next few years, a number of applications were developed with Spring Framework
* Testable but
* Lot of configuration (XML and Java)
* Developing Spring Based application need configuration of a lot of beans!
* Integration with other frameworks need configuration as well!
* In the last few years, focus is moving from monolith applications to microservices. We need to be able to start project quickly. Minimum or Zero start up time
* Framework Setup
* Deployment - Configurability
* Logging, Transaction Management
* Monitoring
* Web Server Configuration

**Spring Boot**

* Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.
* We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss.

**Spring Boot vs Spring MVC:**

There is no relation.

Spring MVC provides decoupled way of developing web applications.

Spring Boot configures Dispatcher Servlet

**Configure Profile:**

Using -Dspring.profiles.active=prod in VM Arguments.

Or

spring.profiles.active=dev in application.properties file

spring.profiles.active=dev (if key is not available it will pick from default properties file)

application.properties (default)

application-dev.properties

application-qa.properties

application-prod.properties



@Profile is used to define bean in specific environment.

To exclude tomcat from spring boot.

<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>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

**Spring Boot Starters:**

spring-boot-devtools: Is used to reflect the code changes automatically without server re start.

**Excluding Resources:**

Certain resources do not necessarily need to trigger a restart when they are changed.

spring.devtools.restart.exclude=static/\*\*,public/\*\*

If you do not like application.properties as the configuration file name, you can switch to another file name by specifying a spring.config.name environment property. You can also refer to an explicit location by using the spring.config.location environment property.

spring.config.name and spring.config.location are used very early to determine which files have to be loaded, so they must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

**Reading properties file key as bean:**

@Component

@ConfigurationProperties("prefix name of property")

public class BasicConfiguration {

}

Autowire this in any class and call getter methods.

<https://github.com/in28minutes/SpringBootForBeginners/blob/master/Step17.md>

spring.application.name=application name

server.port=2018

**What is @SpringBootApplication annotation in spring boot?**

* Many Spring Boot developers always have their main class annotated with @Configuration, @EnableAutoConfiguration and @ComponentScan. Since these annotations are so frequently used together (especially if you follow the best practices above), Spring Boot provides a convenient @SpringBootApplication alternative.
* The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes:
* If your other package hierarchies are below your main app with the @SpringBootApplication annotation, you’re covered by the implicit Component Scan.
* If there are beans/components in other packages that are not sub-packages of the main package, you should manually add them as @ComponentScan({"package1","package2"})

The following are the parameters accepted in the @SpringBootApplication annotation:

**exclude:** Exclude the list of classes from the auto configuration.

**excludeNames:** Exclude the list of fully qualified class names from the auto configuration. This parameter added since spring boot 1.3.0.

**scanBasePackageClasses:** Provide the list of classes that has to be applied for the @ComponentScan.

**scanBasePackages** Provide the list of packages that has to be applied for the @ComponentScan. This parameter added since spring boot 1.3.0.

**How to create war file**

* 1. Change in pom.xml <packaging>war</packaging>
  2. @Override

Protected SpringApplicationBuilder configure(SpringApplicationBuilder app) {

return application.sources(SpringBootWebApplication.class);

}

**Get JSON and XML from service:**

1. http://localhost:2018/hi.json and http://localhost:2018/hi.xml
2. http header accept application/json or application/xml
3. Annotate POJO with @XmlRootElement

# Spring Boot Properties:

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

spring.mvc.view.suffix= .jsp

spring.datasource.url= jdbc:mysql://mysql-container:3306/test

spring.datasource.username= root

spring.datasource.password= root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.jpa.show-sql= true

spring.jpa.hibernate.ddl-auto= update

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

logging.level.org.springframework=DEBUG

server.port=8080 *# Server HTTP port.*

spring.application.name= Test *# Application name.*

server.servlet.context-path= /appName *# Default is ‘/’.*

spring.config.location= *# Default is application.properties*

spring.profiles.active= dev *# Comma-separated list of active profiles.*

spring.jpa.properties.\*= *# Additional native properties to set on the JPA provider.*

***More about @ConfigurationProperties***

<http://www.baeldung.com/configuration-properties-in-spring-boot>

<https://www.boraji.com/spring-boot-configurationproperties-example>

**Difference between @ConfigurationProperties and @Value**

@Value is a core container feature and it does not provide the same features as type-safe Configuration Properties. The table below summarizes the features that are supported by @ConfigurationProperties and @Value:

|  |  |  |
| --- | --- | --- |
| **Feature** | **@ConfigurationProperties** | **@Value** |
| Relaxed binding | Yes | No |
| Meta-data support | Yes | No |
| SpEL evaluation | No | Yes |

If you define a set of configuration keys for your own components, spring boot recommends you to group them in a POJO annotated with @ConfigurationProperties.

**How to run spring boot application through command line?**

java -jar my-sample-application.jar

**Logging using Logback**

<https://springframework.guru/using-logback-spring-boot/>

**Spring Boot – List all Beans loaded in the ApplicationContext**

Spring Boot loads many beans into ApplicationContext on start up depends on the dependencies as per your pom.xml.

You can call applicationContext.getBeanDefinitionNames() method to list all beans loaded into ApplicationContext.

String[] beans = applicationContext.getBeanDefinitionNames();

for (String bean : beans) {

System.out.println(bean);

}

**Spring Framework**

Most important feature of Spring Framework is Dependency Injection. At the core of all Spring Modules is Dependency Injection or IOC Inversion of Control. Loosely coupled applications can be easily unit tested.

**Spring MVC**

Spring MVC Framework provides decoupled way of developing web applications. With simple concepts like Dispatcher Servlet, ModelAndView and View Resolver, it makes it easy to develop web applications.

**Spring Boot**

The problem with spring and Spring MVC is the amount of configuration that is needed. Spring Boot solves this problem through a combination of Auto Configuration and Starter Projects. Spring Boot also provide a few non-functional features to make building production ready applications faster.

**How does spring enable creating production ready applications in quick time?**

Spring Boot aims to enable production ready applications in quick time. Spring Boot provides a few non-functional features out of the box like caching, logging, monitoring and embedded servers.

**spring-boot-starter-actuator** - To use advanced features like monitoring & tracing to your application out of the box

spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat - To pick your specific choice of Embedded Servlet Container

**spring-boot-starter-logging** - For Logging using logback

**spring-boot-starter-cache** - Enabling Spring Framework’s caching support

**Why do we need spring-boot-maven-plugin?**

enable you to package the code as a jar or run the application

**What and Why Embedded Servers?**

Step 1 : Install Java

Step 2 : Install the Web/Application Server (Tomcat/Websphere/Weblogic etc)

Step 3 : Deploy the application war

Step 1: Spring Boot install Java run jar.

**What is Spring Data?**

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services.

To make it simpler, Spring Data provides Abstractions (interfaces) you can use irrespective of underlying data source.

An example is shown below

interface TodoRepository extends CrudRepository<Todo, Long> {

}

You can define a simple repository and use it to insert, update, delete and retrieve todo entities from the database - without writing a lot of code.

### What is Spring Data REST?

Spring Data REST can be used to expose HATEOAS RESTful resources around Spring Data repositories.

An example using JPA is shown below

@RepositoryRestResource(collectionResourceRel = "todos", path = "todos")

public interface TodoRepository

extends PagingAndSortingRepository<Todo, Long> {

Without writing a lot of code, we can expose RESTful API around Spring Data Repositories.

**What is the difference between JPA and Hibernate?**

Short Story

* JPA is a specification/Interface
* Hibernate is one of JPA implementations

When we use JPA, we use the annotation and interfaces from javax.persistence package, without using the hibernate import packages.

We recommend using JPA annotations as we are not tied to Hibernate as implementation. Later (I know - <1% Chance), we can use another JPA implementation.

**How is Hibernate chosen as the default implementation for JPA without any configuration?**

Because of Spring Boot Auto Configuration. This is the dependency we added in

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

The Starter spring-boot-starter-data-jpa has a transitive dependency on Hibernate and JPA.

When Spring Boot sees Hibernate in the class path, it auto configures it as the default JPA Implementation.

**Can you give an example for ReadOnly as true in Transaction management?**

When you read stuff from the database, user details or any other details, you wanna set read only on the transaction so that Hibernate does not need to check for changes to the entities. This is more efficient.

**How to disable specific auto-configuration in spring boot?**

You can use exclude property as shown below to disable specific auto configuration

@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

**What is Hot Swapping in spring boot?**

Reloading the changes without restarting the server is called hot swapping, Modern IDEs (Eclipse, IDEA, etc.) all support hot swapping of byte code,  so if you make a change that doesn’t affect class or method signatures it should reload cleanly with no side effects.

**How do you configure Configure Logback for logging?**

If you put a logback.xml in the root of your classpath it will be picked up from there.

**How do you Configure Log4j for logging?**

Spring Boot supports Log4j 2 for logging configuration if it is on the classpath. If you are using the starters for assembling dependencies that means you have to exclude Logback and then include log4j 2 instead.

**Kafka:**

spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer

spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer

spring.kafka.producer.key-serializer= *#By default uses StringSerializer*

spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.consumer.bootstrap-servers=localhost:9092

spring.kafka.consumer.group-id=myGroup

ProducerFactory or ConsumerFactory

@KafkaTemplate<String, User> template;

template.send("topicname", new User());

@EnableKafka

@KafkaListener(topics="topicname", group="myGroup")

public void consume(String message){

sop(message);

}

We can create 2 ConsumerFactory with different group name and use it.

**Advantages:**

Distributed messaging.

Load balance.