Spring Boot: to create start-up spring application, production ready, with minimal or zero configuration using java/ groovy.

The second class-level annotation is @**EnableAutoConfiguration**. This annotation tells Spring Boot to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

SpringApplication bootstraps our application, starting Spring, which, in turn, starts the auto-configured Tomcat web server. We need to pass Example.class as an argument to the run method to tell SpringApplication which is the primary Spring component. The args array is also passed through to expose any command-line arguments

Executable jars (sometimes called “fat jars”) are archives containing your compiled classes along with all of the jar dependencies that your code needs to run.

To create an executable jar, we need to add the spring-boot-maven-plugin to our pom.xml

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

To run that application, use the java -jar command, as follows:

$ java -jar target/myproject-0.0.1-SNAPSHOT.jar

if you want to get started using Spring and JPA for database access, include the spring-boot-starter-data-jpa dependency in your project.

|  |  |
| --- | --- |
| spring-boot-starter-actuator | Starter for using Spring Boot’s Actuator which provides production ready features to help you monitor and manage your application |

We generally recommend that you locate your main application class in a root package above other classes. The @EnableAutoConfiguration annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items. For example, if you are writing a JPA application, the package of the @EnableAutoConfiguration annotated class is used to search for @Entity items.

Using a root package also lets the @ComponentScan annotation be used without needing to specify a basePackage attribute. You can also use the @SpringBootApplication annotation if your main class is in the root package

You need not put all your @Configuration into a single class. The @Import annotation can be used to import additional configuration classes. Alternatively, you can use @ComponentScan to automatically pick up all Spring components, including @Configuration classes.

If you absolutely must use XML based configuration, we recommend that you still start with a @Configuration class. You can then use an @ImportResource annotation to load XML configuration files.

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, if HSQLDB is on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

You need to opt-in to auto-configuration by adding the @EnableAutoConfiguration or @SpringBootApplication annotations to one of your @Configuration classes.

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](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#using-boot-structuring-your-code) above), Spring Boot provides a convenient @SpringBootApplication alternative.

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes

if you are running your application by using java -jar, you can enable the debug property as follows:

$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug

A SpringApplication attempts to create the right type of ApplicationContext on your behalf. By default, an AnnotationConfigApplicationContext or AnnotationConfigServletWebServerApplicationContext is used, depending on whether you are developing a web application or not.

The algorithm used to determine a “web environment” is fairly simplistic (it is based on the presence of a few classes). If you need to override the default, you can use setWebEnvironment(boolean webEnvironment).

It is also possible to take complete control of the ApplicationContext type that is used by calling setApplicationContextClass(…​).

|  |
| --- |
| [Tip] |
| It is often desirable to call setWebEnvironment(false) when using SpringApplication within a JUnit test. |

If you need to run some specific code once the SpringApplication has started, you can implement the ApplicationRunner or CommandLineRunner interfaces. Both interfaces work in the same way and offer a single run method, which is called just before SpringApplication.run(…​) completes.

If several CommandLineRunner or ApplicationRunner beans are defined that must be called in a specific order, you can additionally implement the org.springframework.core.Ordered interface or use the org.springframework.core.annotation.Order annotation.

Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. Properties are considered in the following order:

1. [Devtools global settings properties](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#using-boot-devtools-globalsettings) on your home directory (~/.spring-boot-devtools.properties when devtools is active).
2. [@TestPropertySource](https://docs.spring.io/spring/docs/5.0.3.BUILD-SNAPSHOT/javadoc-api/org/springframework/test/context/TestPropertySource.html) annotations on your tests.
3. [@SpringBootTest#properties](https://docs.spring.io/spring-boot/docs/2.0.0.BUILD-SNAPSHOT/api/org/springframework/boot/test/context/SpringBootTest.html) annotation attribute on your tests.
4. Command line arguments.

By default, SpringApplication converts any command line option arguments (that is, arguments starting with --, such as --server.port=9000) to a property and adds them to the Spring Environment. As mentioned previously, command line properties always take precedence over other property sources.

If you do not want command line properties to be added to the Environment, you can disable them by using SpringApplication.setAddCommandLineProperties(false).

1. Properties from SPRING\_APPLICATION\_JSON (inline JSON embedded in an environment variable or system property).
2. ServletConfig init parameters.
3. ServletContext init parameters.
4. JNDI attributes from java:comp/env.
5. Java System properties (System.getProperties()).
6. OS environment variables.
7. A RandomValuePropertySource that has properties only in random.\*.
8. [Profile-specific application properties](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-external-config-profile-specific-properties) outside of your packaged jar (application-{profile}.properties and YAML variants).
9. [Profile-specific application properties](https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-external-config-profile-specific-properties) packaged inside your jar (application-{profile}.properties and YAML variants).
10. Application properties outside of your packaged jar (application.properties and YAML variants).
11. Application properties packaged inside your jar (application.properties and YAML variants).

SpringApplication loads properties from application.properties files in the following locations and adds them to the Spring Environment:

1. A /config subdirectory of the current directory
2. The current directory
3. A classpath /config package
4. The classpath root

The list is ordered by precedence

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 (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name:

$ java -jar myproject.jar --spring.config.name=myproject

The following example shows how to specify two locations:

$ java -jar myproject.jar --spring.config.location=classpath:/default.properties,classpath:/override.properties

Config locations are searched in reverse order. By default, the configured locations are classpath:/,classpath:/config/,file:./,file:./config/. The resulting search order is the following:

1. file:./config/
2. file:./
3. classpath:/config/
4. classpath:/

When custom config locations are configured by using spring.config.location, they replace the default locations. For example, if spring.config.location is configured with the value classpath:/custom-config/,file:./custom-config/, the search order becomes the following:

1. file:./custom-config/
2. classpath:custom-config/

Alternatively, when custom config locations are configured by using spring.config.additional-location, they are used in addition to the default locations. Additional locations are searched before the default locations. For example, if additional locations of classpath:/custom-config/,file:./custom-config/ are configured, the search order becomes the following:

1. file:./custom-config/
2. classpath:custom-config/
3. file:./config/
4. file:./
5. classpath:/config/
6. classpath:/

In addition to application.properties files, profile-specific properties can also be defined by using the following naming convention: application-{profile}.properties

you have specified any files in spring.config.location, profile-specific variants of those files are not considered. Use directories in spring.config.location if you want to also use profile-specific properties.

my:

servers:

- dev.example.com

- another.example.com

The preceding example would be transformed into these properties:

my.servers[0]=dev.example.com

my.servers[1]=another.example.com

To bind to properties like that by using the Spring DataBinder utilities (which is what @ConfigurationProperties does), you need to have a property in the target bean of type java.util.List (or Set) and you either need to provide a setter or initialize it with a mutable value. For example, the following example binds to the properties shown previously:

*@ConfigurationProperties(prefix="my")*

public class Config {

private List<String> servers = new ArrayList<String>();

public List<String> getServers() {

return this.servers;

}

}

|  |
| --- |
| [Note] |
|

YAML files cannot be loaded by using the @PropertySource annotation. So, in the case that you need to load values that way, you need to use a properties file.

Using the @Value("${property}") annotation to inject configuration properties can sometimes be cumbersome

Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application,

1. [@PropertySource](https://docs.spring.io/spring/docs/5.0.3.BUILD-SNAPSHOT/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotations on your @Configuration classes.
2. Default properties (specified by setting SpringApplication.setDefaultProperties).

$ java -Dspring.application.json='{"name":"test"}' -jar myapp.jar

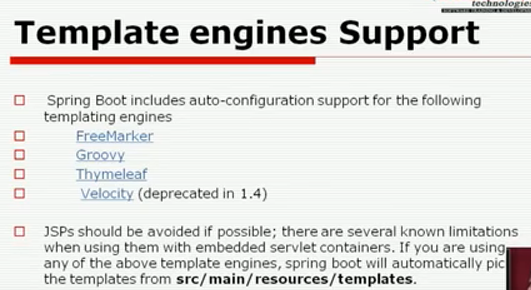
$ java -jar myapp.jar --spring.application.json='{"name":"test"}'

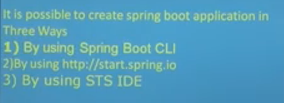
You can also supply the JSON as a JNDI variable, as follows: java:comp/env/spring.application.json.

Reduces development time, increases productivity. Required configurations by default provided by spring boot. Reduces boiler plate coding, annotation, configuration. Easy to integrate with it’s ecosystems like spring jdbc, orm, data, security.. Follows “opinionated default configurations” to reduce developer effort. Has embedded http servers. Provides plugins to work with embedded and in-memory db. Provides some starter projects and avoids xml config completely. Provides cli to run applications from command prompt. Plugins to develop and test app using build tools like maven, gradle.

Limitations:

1. Tough to convert legacy app to spring boot.





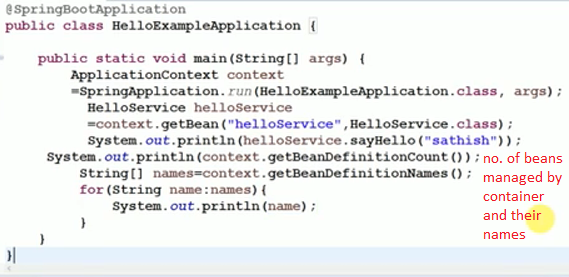
Spring-boot-starter-web: reqd configuration for springdispatcher done by spring boot.

SpringBootApplication: Annotated with other annotations:





SpringApplication.run returns spring container.



Spring-boot-starter-actuator: Provides production ready features to help monitor and manage the application.

Spring-boot-starter-jetty: jetty as embedded server

Rapid application Development

create stand alone spring based application

Spring Boot does not generate code and there is absolutely no requirement for XML configuration.

Embed Tomcat, Jetty or Undertow directly. You don't need to deploy WAR files

It provides opinionated 'starter' POMs to simplify your Maven configuration.

It provides production-ready features such as metrics, health checks and externalized configuration

Spring: framework to build java EE applications. Transaction management, programming and configuration model (annotations), infrastructure support (connect to db)

Problems with spring: huge framework, multiple set-up steps, multiple configuration & build-deploy steps

SpringApplication

New maven project: simple java archetype

It is a class which provides the convenient way to bootstrap a spring application which can be started from main method. You can call start your application just by calling a static run() method.

1. Sets up default configuration
2. Starts spring application context
3. Performs class-path scan
4. Starts tomcat server, deploys application.

@SpringBootApplication

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

1. SpringApplication.run(className.**class**, args);
2. }
3. **<parent>**
4. **<groupId>**org.springframework.boot**</groupId>**
5. **<artifactId>**spring-boot-starter-parent**</artifactId>**
6. **<version>**1.4.2.RELEASE**</version>**
7. **</parent>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**org.springframework.boot**</groupId>**
11. **<artifactId>**spring-boot-starter-web**</artifactId>**
12. **</dependency>**
13. **</dependencies>**

<properties>

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

</properties>

<plugin>

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

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

</plugin>

## ­--------------------------------------------------------------------------

## Simple Project 1:

## ---------------------------------------------------------------------------

## 

## Bydefault spring boot serves static web content from the /static or /public or /resources or /META-INF/resources folder in classpath. (css, js)

## Spring boot loads properties from application.properties and adds them to spring environment.

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

## spring.mvc.view.suffix= .jsp

## spring.mvc.static-path-pattern= /resources/\*\*

## 

*@RestController*

*@EnableAutoConfiguration*

**public** **class** Example {

*@RequestMapping("/")*

String home() {

**return** "Hello World!";

}

## ---------------------------------------------------------------------

## @SpringBootApplication

It is composed of:

1. @SpringBootConfiguration
2. @EnableAutoConfiguration
3. @ComponentScan(excludeFilters = @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class))

#### @SpringBootConfiguration

|  |
| --- |
| @Configuration  public @interface SpringBootConfiguration  {      //more code  } |

1. This annotation adds @Configuration annotation to class which **mark the class a source of bean definitions for the application context.**

#### @EnableAutoConfiguration

This tells spring boot to auto configure important bean definitions based on added dependencies in pom.xml by start adding beans based on classpath settings, other beans, and various property settings.

#### @ComponentScan

This annotation tells spring boot to scan base package, find other beans/components and configure them as well.

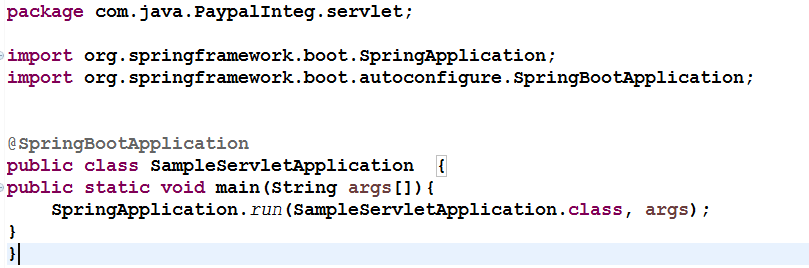
Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

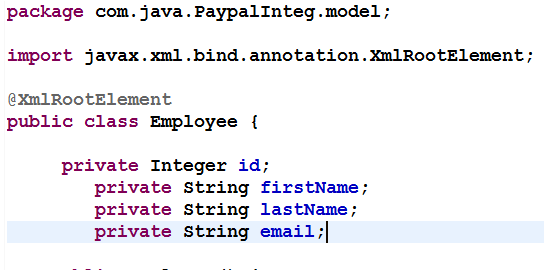
**Place @EnableAutoConfiguration in a root package so that all sub-packages and classes can be searched.**

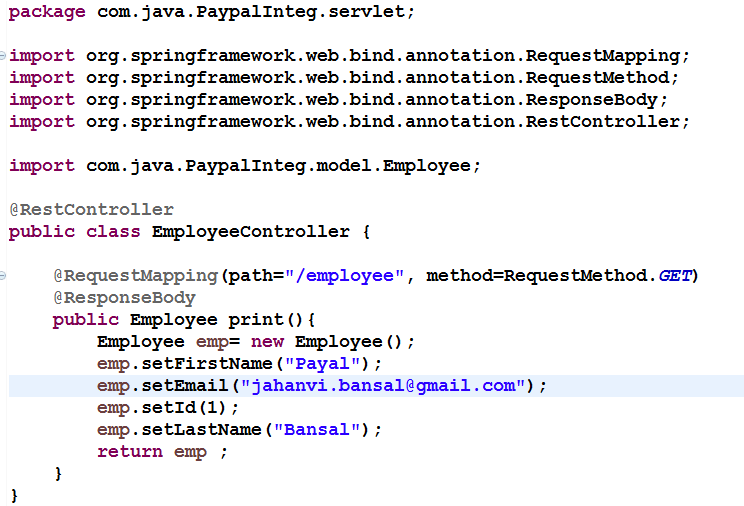
**---------------------------------------------------------------------------------------------------------**

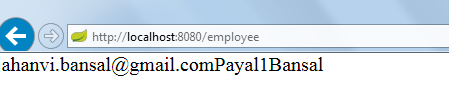
**Simple Project2:**

**---------------------------------------------------------------------------------------------------------**

****

****

****

****

**Note: Main class should be in parent/same package as Controller class else do component scan**

**Should not mention context root in url**

**DispatcherServlet initialized using autoconfigure.: error handling,**

**Spring-boot-autoconfigure.jar**

**Application.properties in resources folder.**

**--------------------------------------------------------------------------------------------------------**

**Spring Boot +Hiberrnate:**

The data source properties starting with spring.datasource.\* will automatically be read by spring boot JPA. To change the hibernate properties we will use prefix spring.jpa.properties.\* with hibernate property name

**Spring + Bootstrap**

Maven project -> web-app archetype.



### JSP Support

We want to use JSP as the view. Tomcat is the default embedded servlet container for Spring Boot Starter Web. To enable support for JSPs, we would need to add a dependency on tomcat-embed-jasper.

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>