



Cos' è Spring Boot

```
/**
```

- * Spring Boot makes it easy to create stand-
 - * alone, production-grade Spring based
 - * Applications that you can "just run".
- ```
*/
```

*Spring.io*

# Cos' è Spring Boot

/\*\*

- \* 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
- \* minimal Spring configuration.

\*/

*Spring.io*

# Spring Boot Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration
- Integrate with a huge number of other components out of the box like:
  - Consul
  - Netflix Eureka
  - Admin console
  - ecc

# My first Spring Boot application

- Create a maven project and add spring boot starter as dependencies

```
...
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter</artifactId>
</dependency>
...
```

- and spring boot maven plugin

```
...
<plugin>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
...
```

# My first Spring Boot application

- Create the main class under defined package in pom.xml
- `import org.springframework.boot.autoconfigure.SpringBootApplication`
- Annotate the class with `@SpringBootApplication`
- Under main method, initialize Spring boot application

```
SpringApplication.run(DemoCleanApplication.class, args);
```

- Passing as first argument the Main Class and as second argument the arguments of main method.
- Run the application using maven goal `mvn spring-boot:run`

THAT'S IT!!!

# My first Spring Boot application

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoCleanApplication {

 public static void main(String[] args) {
 SpringApplication.run(DemoCleanApplication.class, args);
 }

}
```

# What @SpringBootApplication do?

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

- @EnableAutoConfiguration: enable [Spring Boot's auto-configuration mechanism](#)
- @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](#))
- @Configuration: allow to register extra beans in the context or import additional configuration classes

@SpringBootApplication also provides aliases to customize the attributes of @EnableAutoConfiguration and @ComponentScan



# My first Spring Boot application

We have created a **Spring Boot** Application with **zero** configurations.

At this point the application has started, but it do not anythings.

Let's add some components...

# My first Spring Boot component

- Create under the base package a new class: HelloWorldController.java
- `import org.springframework.stereotype.Component`
- Annotate the class with `@Component`
- Add default constructor
- Inside the default construct add: `System.out.println("Hello Worl!");`
- Run the application with: `mvn spring-boot:run`
- On the console at the startup you can find the following message: `Hello Worl!`

THAT'S IT!!!

# My first Spring Boot component

```
package com.example.democlean;

import org.springframework.stereotype.Component;

@Component
class HelloWorldComponent {

 HelloWorldComponent() {
 System.out.println("Hello Worl!");
 }

}
```

# My first Spring Boot: Web

- Let's create a web controller by adding the spring boot starter web dependency:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- Create a new class under base package: HelloWorldController.java

Import spring web annotations packages:

```
import org.springframework.web.bind.annotation.RestController
import org.springframework.web.bind.annotation.RequestMapping;
```

- Annotate the class with **@RestController**
- Create a public method that return a string «Hello World!!!» annotated with **@RequestMapping("/")**
- Run the application, open the web browser and visit: <http://localhost:8080> to see the message "Hello world!!!".

# My first Spring Boot: Web

```
package com.example.democlean;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController {

 @RequestMapping("/")
 public String sayHello() {
 return "Hello world!!!";
 }
}
```

# Spring Boot: Configurations

We have not written any configurations so far.

Let's define our "Hello World!!!" string inside a variable.

- Create "**resources**" folder under src/main
- Add "**application.properties**" file
- Open the file and add "**helloController.message=Hello World**"
- Open HelloWorldController.java and add the import of Value annotation  
`import org.springframework.beans.factory.annotation.Value;`
- Add a String property annotate with `@Value("${helloController.message}")`
- Replace returned value inside the method sayHello with the defined property.
- Run the application and go to <http://localhost:8080>

# Spring Boot: Configurations

```
package com.example.democlean;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController {

 @Value("${helloController.message}")
 private String message;

 @RequestMapping("/")
 public String sayHello() {
 return message;
 }
}
```

# Spring Boot: Configurations

We have defined our first custom property and spring boot will automatically load application.properties and give us access to all defined properties.

You can also override library properties like:

- `server.port=8090`
- `server.servlet.context-path=/api`



# Spring Boot: Actuator

Spring Boot includes a number of additional features to help you monitor and manage your application when you push it to production. You can choose to manage and monitor your application by using HTTP endpoints or with JMX. Auditing, health, and metrics gathering can also be automatically applied to your application.

The [spring-boot-actuator](#) module provides all of Spring Boot's production-ready features. The recommended way to enable the features is to add a dependency on the spring-boot-starter-actuator 'Starter'.

## Definition of Actuator

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.

# Spring Boot: Actuator

To add the actuator to a Maven based project, add the following 'Starter' dependency:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

# Spring Boot: Actuator Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.

Each individual endpoint can be [enabled or disabled](#) and [exposed \(made remotely accessible\) over HTTP or JMX](#). An endpoint is considered to be available when it is both enabled and exposed. The built-in endpoints will only be auto-configured when they are available. Most applications choose exposure 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**.

## Enabling Endpoints:

By default, all endpoints except for shutdown are enabled. To configure the enablement of an endpoint, use its **management.endpoint.<id>.enabled** property. The following example enables the shutdown endpoint:

```
management.endpoint.shutdown.enabled=true
```

# Spring Boot: Actuator Endpoints

If you prefer endpoint enablement to be opt-in rather than opt-out, set the `management.endpoints.enabled-by-default` property to `false` and use individual endpoint enabled properties to opt back in. The following example enables the info endpoint and disables all other endpoints:

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

Disabled endpoints are removed entirely from the application context. If you want to change only the technologies over which an endpoint is exposed, use the [include and exclude properties](#) instead.

# Spring Boot: Actuator Endpoints

## Exposing Endpoints

Since Endpoints may contain sensitive information, careful consideration should be given about when to expose them. The following table shows the default exposure for the built-in endpoints:

ID	JMX	Web
auditevents	Yes	No
beans	Yes	No
caches	Yes	No
conditions	Yes	No
configprops	Yes	No
env	Yes	No
flyway	Yes	No
health	Yes	Yes
heapdump	N/A	No
httptrace	Yes	No
info	Yes	Yes
integrationgraph	Yes	No
jolokia	N/A	No
logfile	N/A	No
loggers	Yes	No
liquibase	Yes	No
metrics	Yes	No
mappings	Yes	No
prometheus	N/A	No
scheduledtasks	Yes	No
sessions	Yes	No
shutdown	Yes	No
threaddump	Yes	No

# Spring Boot: Actuator Endpoints

To change which endpoints are exposed, use the following technology-specific include and exclude properties:

Property	Default
management.endpoints.jmx.exposure.exclude	
management.endpoints.jmx.exposure.include	*
management.endpoints.web.exposure.exclude	
management.endpoints.web.exposure.include	info, health

The include property lists the IDs of the endpoints that are exposed. The exclude property lists the IDs of the endpoints that should not be exposed. The exclude property takes precedence over the include property.

Both include and exclude properties can be configured with a list of endpoint IDs.

For example, to stop exposing all endpoints over JMX and only expose the health and info endpoints, use the following property:

```
management.endpoints.jmx.exposure.include=health,info
```

# Spring Boot: Actuator Endpoints

## Securing HTTP Endpoints

You should take care to secure HTTP endpoints in the same way that you would any other sensitive URL. If Spring Security is present, endpoints are secured by default using Spring Security's content-negotiation strategy. If you wish to configure custom security for HTTP endpoints, for example, only allow users with a certain role to access them, Spring Boot provides some convenient RequestMatcher objects that can be used in combination with Spring Security.

A typical Spring Security configuration might look something like the following example:

```
@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();
 }
}
```

# Spring Boot: Actuator Endpoints

## Monitoring and Management over HTTP

If you are developing a web application, Spring Boot Actuator auto-configures all enabled endpoints to be exposed over HTTP. The default convention is to use the id of the endpoint with a prefix of /actuator as the URL path. For example, health is exposed as /actuator/health.

```
{
 "status": "UP"
}
```



# Spring Boot: The Executable Jar Format

The spring-boot-loader module lets Spring Boot support executable jar and war files. If you use the Maven plugin or the Gradle plugin, executable jars are automatically generated, and you generally do not need to know the details of how they work.

If you need to create executable jars from a different build system or if you are just curious about the underlying technology, this appendix provides some background.

# Spring Boot: The Executable Jar Format

## The Executable Jar File Structure

```
example.jar
|
+-META-INF
| +-MANIFEST.MF
+-org
| +-springframework
| +-boot
| +-loader
| +-<spring boot loader classes>
+-BOOT-INF
+-classes
| +-mycompany
| +-project
| +-YourClasses.class
+-lib
+-dependency1.jar
+-dependency2.jar
```

# Spring Boot: The Executable War File Structure

```
example.war
|
|--META-INF
| |--MANIFEST.MF
|--org
| |--springframework
| |--boot
| |--loader
| |--<spring boot loader classes>
|--WEB-INF
|--classes
| |--com
| |--mycompany
| |--project
| |--YourClasses.class
|--lib
| |--dependency1.jar
| |--dependency2.jar
|--lib-provided
|--servlet-api.jar
|--dependency3.jar
```

# Codecentric's Spring Boot Admin

This community project provides an admin interface for [Spring Boot](#)® applications. Monitoring Python applications is available using [Pyctuator](#).

Spring Boot Admin provides the following features for registered applications:

- Show health status
- Show details, like
  - JVM & memory metrics
  - micrometer.io metrics
  - Datasource metrics
  - Cache metrics
- Show build-info number
- Follow and download logfile
- View jvm system- & environment-properties
- View Spring Boot Configuration Properties
- Support for Spring Cloud's postable /env- &/refresh-endpoint
- Easy loglevel management
- Interact with JMX-beans
- View thread dump
- View http-traces
- View auditevents
- View http-endpoints
- View scheduled tasks
- View and delete active sessions (using spring-session)
- View Flyway / Liquibase database migrations
- Download heapdump
- Notification on status change (via e-mail, Slack, Hipchat, ...)
- Event journal of status changes (non persistent)

<https://github.com/codecentric/spring-boot-admin>

# Codecentric's Spring Boot Admin

First, you need to setup your server. To do this just setup a simple boot project (using [start.spring.io](https://start.spring.io)). As Spring Boot Admin Server is capable of running as servlet or webflux application, you need to decide on this and add the according Spring Boot Starter. In this example we are using the servlet web starter.

```
<dependency>
 <groupId>de.codecentric</groupId>
 <artifactId>spring-boot-admin-starter-server</artifactId>
 <version>2.3.0</version>
</dependency>
```

# Codecentric's Spring Boot Admin

- Create a configuration class to enable admin server: AdminServerConfig.java
- Import `import org.springframework.context.annotation.Configuration;`
- Annotate the class with `@Configuration`.
  - Indicates that a class declares one or more [@Bean](#) methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

```
@Configuration
@EnableAdminServer
public class AdminServerConfig {
}
```

# Codecentric's Spring Boot Admin Client

First add required dependency:

```
<dependency>
 <groupId>de.codecentric</groupId>
 <artifactId>spring-boot-admin-starter-client</artifactId>
 <version>2.3.0</version>
</dependency>
```

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

```
<dependency>
 <groupId>org.jolokia</groupId>
 <artifactId>jolokia-core</artifactId>
</dependency>
```

# Codecentric's Spring Boot Admin Client

Add some basic configurations:

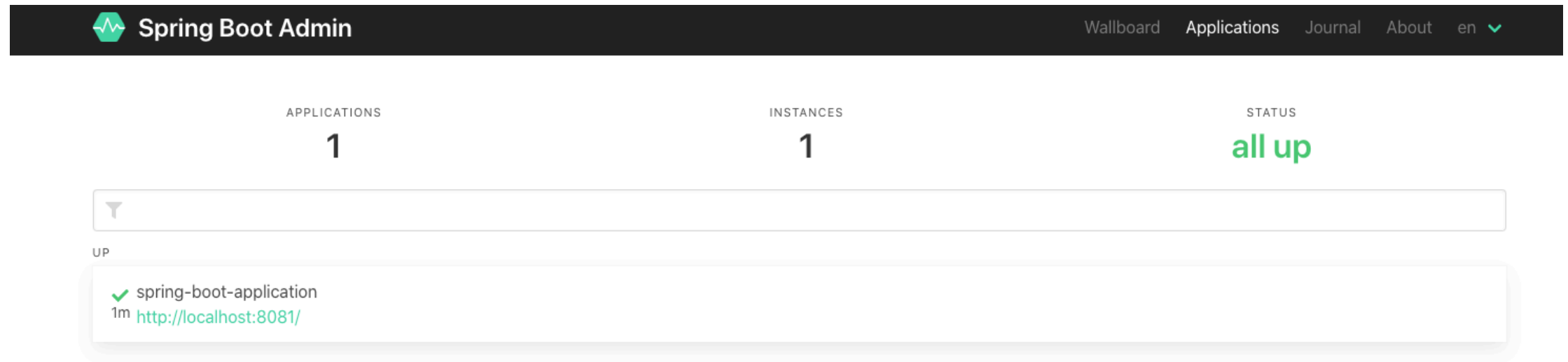
```
server.port=8081 //Change the port to run both server:8080 and
client:8081 on localhost
```

```
spring.boot.admin.client.url=http://localhost:8080
management.endpoints.web.exposure.include=*
```



# Codecentric's Spring Boot Admin Client

- Run server and client
- Open web browser and go to <http://localhost:8080> (admin server url)



The screenshot displays the Spring Boot Admin web interface. At the top, a dark navigation bar contains the 'Spring Boot Admin' logo and title on the left, and links for 'Wallboard', 'Applications', 'Journal', 'About', and 'en' with a dropdown arrow on the right. Below the navigation bar, three summary statistics are shown: 'APPLICATIONS' with a value of '1', 'INSTANCES' with a value of '1', and 'STATUS' with the text 'all up' in green. A search bar with a funnel icon is positioned below these statistics. Under the search bar, the word 'UP' is displayed. A single application entry is listed, showing a green checkmark, the name 'spring-boot-application', a '1m' refresh indicator, and the URL 'http://localhost:8081/'.

# END