**Spring Boot 参考指南**

1.5.8.RELEASE

Phillip Webb , Dave Syer , Josh Long , Stéphane Nicoll , Rob Winch , Andy Wilkinson , Marcel Overdijk , Christian Dupuis , Sébastien Deleuze , Michael Simons

Copyright © 2012-2017

本文件的副本可供你自己使用，并分发给别人，但你不收取此类副本的人户费用，并进一步规定没分副本均包含本版权声明，不论是以书面形式还是以电子方式分发。

**目录**

1. [Spring Boot 文档](#_bookmark0) [1](#_bookmark0)
   1. 关于文档 [2](#_bookmark1)
   2. 获取帮助 [3](#_bookmark2)
   3. 第一步 [4](#_bookmark3)
   4. [使用 Spring Boot](#_bookmark4) [5](#_bookmark4)
   5. [了解 Spring Boot 功能](#_bookmark5) [6](#_bookmark5)
   6. 生产中使用 [7](#_bookmark6)
   7. 高级主题 [8](#_bookmark7)
2. 开始 [9](#_bookmark8)
3. [介绍 Spring Boot](#_bookmark9) [10](#_bookmark9)
4. 系统要求 [11](#_bookmark10)
   1. [Servlet 容器](#_bookmark11) [11](#_bookmark11)
5. [安装 Spring Boot](#_bookmark12) [12](#_bookmark12)
   1. Java开发人员安装说明 [12](#_bookmark13)

[Maven 安装](#_bookmark14) [12](#_bookmark14)

[Gradle 安装](#_bookmark15) [13](#_bookmark15)

* 1. [安装 Spring Boot CLI](#_bookmark16) [14](#_bookmark16)

手动安装 [14](#_bookmark17)

[安装与 SDKMAN!](#_bookmark18) [14](#_bookmark18)

[OSX 自带程序安装](#_bookmark19) [15](#_bookmark19)

[MacPorts 安装](#_bookmark20) [15](#_bookmark20)

命令行完成 [15](#_bookmark21)

[快速开始 Spring CLI 例子](#_bookmark22) [15](#_bookmark22)

* 1. [从较早版本的Spring Boot升级](#_bookmark23) [16](#_bookmark23)

1. [开发你的第一个 Spring Boot 应用](#_bookmark24) [17](#_bookmark24)
   1. [创建 POM](#_bookmark25) [17](#_bookmark25)
   2. [添加 dependencies](#_bookmark26) [18](#_bookmark26)
   3. 写代码 [18](#_bookmark27)

[@RestController 和 @RequestMapping 注解](#_bookmark28) [19](#_bookmark28)

[@EnableAutoConfiguration 注解](#_bookmark29) [19](#_bookmark29)

[“main” 方法](#_bookmark30) [19](#_bookmark30)

* 1. 运行示例 [19](#_bookmark31)
  2. [创建可执行的 jar](#_bookmark32) [20](#_bookmark32)

1. [What to read next](#_bookmark33) [22](#_bookmark33)
2. [使用 Spring Boot](#_bookmark34) [23](#_bookmark34)
3. 构建系统 [24](#_bookmark35)
   1. 依赖管理 [24](#_bookmark36)
   2. [Maven](#_bookmark37) [24](#_bookmark37)

[继承 starter parent](#_bookmark38) [24](#_bookmark38)

[没有父POM使用Spring Boot](#_bookmark39)  [25](#_bookmark39)

[变更Java 版本](#_bookmark40) [26](#_bookmark40)

[使用 Maven plugin Spring Boot](#_bookmark41)  [26](#_bookmark41)

* 1. [Gradle](#_bookmark42) [26](#_bookmark42)
  2. [Ant](#_bookmark43) [27](#_bookmark43)
  3. [Starters](#_bookmark44) [28](#_bookmark44)

1. [结构化你的代码](#_bookmark50) [33](#_bookmark50)
   1. [使用默认的包](#_bookmark51) [33](#_bookmark51)
   2. [定位 main 程序类](#_bookmark52) [33](#_bookmark52)
2. [配置 classes](#_bookmark53) [35](#_bookmark53)
   1. [导入外部配置 classes](#_bookmark54) [35](#_bookmark54)
   2. [导入 XML 配置](#_bookmark55) [35](#_bookmark55)
3. 自动装配 [36](#_bookmark56)
   1. [逐渐替换 auto-configuration](#_bookmark57) [36](#_bookmark57)
   2. [禁用特定的 auto-configuration](#_bookmark58) [36](#_bookmark58)
4. [Spring Beans 和 dependency injection](#_bookmark59) [37](#_bookmark59)
5. [使用 @SpringBootApplication 注解](#_bookmark60) [38](#_bookmark60)
6. [运行 application](#_bookmark61) [39](#_bookmark61)
   1. [从 IDE](#_bookmark62) 运行 [39](#_bookmark62)
   2. [打包运行](#_bookmark63)  [39](#_bookmark63)
   3. [使用 Maven plugin](#_bookmark64) [39](#_bookmark64)
   4. [使用 Gradle plugin](#_bookmark65) [40](#_bookmark65)
   5. [Hot swapping](#_bookmark66) [40](#_bookmark66)
7. 开发者工具 [41](#_bookmark67)
   1. 默认属性设置 [41](#_bookmark68)
   2. 自动重新启动 [42](#_bookmark69)

[Excluding resources](#_bookmark71) [43](#_bookmark71)

[监控外部 paths](#_bookmark72) [43](#_bookmark72)

禁止重启 [43](#_bookmark73)

[使用 trigger 文件](#_bookmark74) [44](#_bookmark74)

[自定义重新启动 classloader](#_bookmark75) [44](#_bookmark75)

已知限制 [44](#_bookmark76)

* 1. [LiveReload](#_bookmark77) [45](#_bookmark77)
  2. 全局设置 [45](#_bookmark78)
  3. 远程应用程序 [45](#_bookmark79)

[运行远程客户端应用程序](#_bookmark80) [46](#_bookmark80)

远程更新 [47](#_bookmark81)

远程调试 [47](#_bookmark82)

1. [打包应用程序用户生产](#_bookmark83) [49](#_bookmark83)
2. 下一步读的内容 [50](#_bookmark84)
3. [Spring Boot 特性](#_bookmark85) [51](#_bookmark85)
4. [SpringApplication](#_bookmark86) [52](#_bookmark86)
   1. 启动故障 [52](#_bookmark87)
   2. [自定义 Banner](#_bookmark88) [53](#_bookmark88)
   3. [自定义 SpringApplication](#_bookmark89) [54](#_bookmark89)
   4. [流利生成 API](#_bookmark90) [54](#_bookmark90)
   5. [应用 events and listeners](#_bookmark91) [55](#_bookmark91)
   6. [Web 环境](#_bookmark92) [55](#_bookmark92)
   7. 应用程序访问参数 [56](#_bookmark93)
   8. [使用 ApplicationRunner 或 CommandLineRunner](#_bookmark94) [56](#_bookmark94)
   9. 退出应用 [57](#_bookmark95)
   10. 管理功能 [57](#_bookmark96)
5. 外部化部署 [59](#_bookmark97)
   1. 配置随机值 [60](#_bookmark98)
   2. 命令行访问属性 [60](#_bookmark99)
   3. 应用属性文件 [61](#_bookmark100)
   4. [Profile-specific properties](#_bookmark101) [62](#_bookmark101)
   5. [Placeholders in properties](#_bookmark102) [63](#_bookmark102)
   6. [使用 YAML 代替 Properties](#_bookmark103) [63](#_bookmark103)

[加载 YAML](#_bookmark104) [63](#_bookmark104)

[Exposing YAML as properties in the Spring Environment](#_bookmark105) [64](#_bookmark105)

[Multi-profile YAML 文档](#_bookmark106) [64](#_bookmark106)

[YAML 缺陷](#_bookmark107) [65](#_bookmark107)

[Merging YAML lists](#_bookmark108) [65](#_bookmark108)

* 1. [Type-safe 配置文件](#_bookmark109) [66](#_bookmark109)

[Third-party 配置](#_bookmark110) [69](#_bookmark110)

[Relaxed binding](#_bookmark111) [69](#_bookmark111)

[Properties 转换](#_bookmark112) [70](#_bookmark112)

[@ConfigurationProperties 注解](#_bookmark113) [70](#_bookmark113)

[@ConfigurationProperties vs. @Value](#_bookmark114) [71](#_bookmark114)

1. [Profiles](#_bookmark115) [72](#_bookmark115)
   1. [Adding active profiles](#_bookmark116) [72](#_bookmark116)
   2. [Programmatically setting profiles](#_bookmark117) [72](#_bookmark117)
   3. [Profile-specific configuration files](#_bookmark118) [73](#_bookmark118)
2. [Logging](#_bookmark119) [74](#_bookmark119)
   1. [Log format](#_bookmark120) [74](#_bookmark120)
   2. [Console output](#_bookmark121) [74](#_bookmark121)

[Color-coded output](#_bookmark122) [75](#_bookmark122)

* 1. [File output](#_bookmark123) [76](#_bookmark123)
  2. [Log Levels](#_bookmark124) [76](#_bookmark124)
  3. 自定义日志配置文件 [77](#_bookmark125)
  4. [Logback extensions](#_bookmark126) [78](#_bookmark126)

[Profile-specific configuration](#_bookmark127) [79](#_bookmark127)

[Environment properties](#_bookmark128) [79](#_bookmark128)

1. [Developing web applications](#_bookmark129) [80](#_bookmark129)
   1. [The ‘Spring Web MVC framework’](#_bookmark130) [80](#_bookmark130)

[Spring MVC auto-configuration](#_bookmark131) [80](#_bookmark131)

[HttpMessageConverters](#_bookmark132) [81](#_bookmark132)

[Custom JSON Serializers and Deserializers](#_bookmark133) [81](#_bookmark133)

[MessageCodesResolver](#_bookmark134) [82](#_bookmark134)

[Static Content](#_bookmark135) [82](#_bookmark135)

[Custom Favicon](#_bookmark136) [84](#_bookmark136)

[ConfigurableWebBindingInitializer](#_bookmark137) [84](#_bookmark137)

[Template engines](#_bookmark138) [84](#_bookmark138)

[Error Handling](#_bookmark139) [84](#_bookmark139)

[Custom error pages](#_bookmark140) [85](#_bookmark140)

[Mapping error pages outside of Spring MVC](#_bookmark141) [86](#_bookmark141)

[Error Handling on WebSphere Application Server](#_bookmark142) [86](#_bookmark142)

[Spring HATEOAS](#_bookmark143) [87](#_bookmark143)

[CORS support](#_bookmark144) [87](#_bookmark144)

* 1. [JAX-RS and Jersey](#_bookmark145) [87](#_bookmark145)
  2. [Embedded servlet container support](#_bookmark146) [88](#_bookmark146)

[Servlets, Filters, and listeners](#_bookmark147) [89](#_bookmark147)

[Registering Servlets, Filters, and listeners as Spring beans](#_bookmark148) [89](#_bookmark148)

[Servlet Context Initialization](#_bookmark149) [89](#_bookmark149)

[Scanning for Servlets, Filters, and listeners](#_bookmark150) [89](#_bookmark150)

[The EmbeddedWebApplicationContext](#_bookmark151) [89](#_bookmark151)

[Customizing embedded servlet containers](#_bookmark152) [90](#_bookmark152)

[Programmatic customization](#_bookmark153) [90](#_bookmark153)

[Customizing ConfigurableEmbeddedServletContainer directly](#_bookmark154) [91](#_bookmark154)

[JSP limitations](#_bookmark155) [91](#_bookmark155)

1. [Security](#_bookmark156) [92](#_bookmark156)
   1. [OAuth2](#_bookmark157) [93](#_bookmark157)

[Authorization Server](#_bookmark158) [93](#_bookmark158)

[Resource Server](#_bookmark159) [93](#_bookmark159)

* 1. [Token Type in User Info](#_bookmark160) [94](#_bookmark160)
  2. [Customizing the User Info RestTemplate](#_bookmark161) [94](#_bookmark161)

[Client](#_bookmark162) [95](#_bookmark162)

[Single Sign On](#_bookmark163) [96](#_bookmark163)

* 1. [Actuator Security](#_bookmark164) [97](#_bookmark164)

1. [Working with SQL databases](#_bookmark165) [98](#_bookmark165)
   1. [Configure a DataSource](#_bookmark166) [98](#_bookmark166)

[Embedded Database Support](#_bookmark167) [98](#_bookmark167)

[Connection to a production database](#_bookmark168) [99](#_bookmark168)

[Connection to a JNDI DataSource](#_bookmark169) [100](#_bookmark169)

* 1. [Using JdbcTemplate](#_bookmark170) [100](#_bookmark170)
  2. [JPA and ‘Spring Data’](#_bookmark171) [101](#_bookmark171)

[Entity Classes](#_bookmark172) [101](#_bookmark172)

[Spring Data JPA Repositories](#_bookmark173) [102](#_bookmark173)

[Creating and dropping JPA databases](#_bookmark174) [103](#_bookmark174)

[Open EntityManager in View](#_bookmark175) [103](#_bookmark175)

* 1. [Using H2’s web console](#_bookmark176) [103](#_bookmark176)

[Changing the H2 console’s path](#_bookmark177) [103](#_bookmark177)

[Securing the H2 console](#_bookmark178) [104](#_bookmark178)

* 1. [Using jOOQ](#_bookmark179) [104](#_bookmark179)

[Code Generation](#_bookmark180) [104](#_bookmark180)

[Using DSLContext](#_bookmark181) [104](#_bookmark181)

[Customizing jOOQ](#_bookmark182) [105](#_bookmark182)

1. [Working with NoSQL technologies](#_bookmark183) [106](#_bookmark183)
   1. [Redis](#_bookmark184) [106](#_bookmark184)

[Connecting to Redis](#_bookmark185) [106](#_bookmark185)

* 1. [MongoDB](#_bookmark186) [106](#_bookmark186)

[Connecting to a MongoDB database](#_bookmark187) [106](#_bookmark187)

[MongoTemplate](#_bookmark188) [107](#_bookmark188)

[Spring Data MongoDB repositories](#_bookmark189) [108](#_bookmark189)

[Embedded Mongo](#_bookmark190) [108](#_bookmark190)

[30.3. Neo4j](#_bookmark191) [109](#_bookmark191)

[Connecting to a Neo4j database](#_bookmark192) [109](#_bookmark192)

[Using the embedded mode](#_bookmark193) [109](#_bookmark193)

[Neo4jSession](#_bookmark194) [109](#_bookmark194)

[Spring Data Neo4j repositories](#_bookmark195) [110](#_bookmark195)

[Repository example](#_bookmark196) [110](#_bookmark196)

* 1. [Gemfire](#_bookmark197) [110](#_bookmark197)
  2. [Solr](#_bookmark198) [110](#_bookmark198)

[Connecting to Solr](#_bookmark199) [111](#_bookmark199)

[Spring Data Solr repositories](#_bookmark200) [111](#_bookmark200)

* 1. [Elasticsearch](#_bookmark201) [111](#_bookmark201)

[Connecting to Elasticsearch using Jest](#_bookmark202) [111](#_bookmark202)

[Connecting to Elasticsearch using Spring Data](#_bookmark203) [112](#_bookmark203)

[Spring Data Elasticsearch repositories](#_bookmark204) [112](#_bookmark204)

* 1. [Cassandra](#_bookmark205) [112](#_bookmark205)

[Connecting to Cassandra](#_bookmark206) [113](#_bookmark206)

[Spring Data Cassandra repositories](#_bookmark207) [113](#_bookmark207)

* 1. [Couchbase](#_bookmark208) [113](#_bookmark208)

[Connecting to Couchbase](#_bookmark209) [113](#_bookmark209)

[Spring Data Couchbase repositories](#_bookmark210) [114](#_bookmark210)

* 1. [LDAP](#_bookmark211) [115](#_bookmark211)

[Connecting to an LDAP server](#_bookmark212) [115](#_bookmark212)

[Spring Data LDAP repositories](#_bookmark213) [115](#_bookmark213)

[Embedded in-memory LDAP server](#_bookmark214) [115](#_bookmark214)

1. [Caching](#_bookmark215) [117](#_bookmark215)
   1. [Supported cache providers](#_bookmark216) [118](#_bookmark216)

[Generic](#_bookmark217) [119](#_bookmark217)

[JCache (JSR-107)](#_bookmark218) [119](#_bookmark218)

[EhCache 2.x](#_bookmark219) [120](#_bookmark219)

[Hazelcast](#_bookmark220) [120](#_bookmark220)

[Infinispan](#_bookmark221) [120](#_bookmark221)

[Couchbase](#_bookmark222) [120](#_bookmark222)

[Redis](#_bookmark223) [121](#_bookmark223)

[Caffeine](#_bookmark224) [121](#_bookmark224)

[Guava (deprecated)](#_bookmark225) [122](#_bookmark225)

[Simple](#_bookmark226) [122](#_bookmark226)

[None](#_bookmark227) [122](#_bookmark227)

1. [Messaging](#_bookmark228) [123](#_bookmark228)

[32.1. JMS](#_bookmark229) [123](#_bookmark229)

[ActiveMQ support](#_bookmark230) [123](#_bookmark230)

[Artemis support](#_bookmark231) [124](#_bookmark231)

[Using a JNDI ConnectionFactory](#_bookmark232) [124](#_bookmark232)

[Sending a message](#_bookmark233) [124](#_bookmark233)

[Receiving a message](#_bookmark234) [125](#_bookmark234)

* 1. [AMQP](#_bookmark235) [126](#_bookmark235)

[RabbitMQ support](#_bookmark236) [126](#_bookmark236)

[Sending a message](#_bookmark237) [126](#_bookmark237)

[Receiving a message](#_bookmark238) [127](#_bookmark238)

* 1. [Apache Kafka Support](#_bookmark239) [128](#_bookmark239)

[Sending a Message](#_bookmark240) [128](#_bookmark240)

[Receiving a Message](#_bookmark241) [129](#_bookmark241)

[Additional Kafka Properties](#_bookmark242) [129](#_bookmark242)

1. [Calling REST services](#_bookmark243) [131](#_bookmark243)
   1. [RestTemplate customization](#_bookmark244) [131](#_bookmark244)
2. [Validation](#_bookmark245) [133](#_bookmark245)
3. [Sending email](#_bookmark246) [134](#_bookmark246)
4. [Distributed Transactions with JTA](#_bookmark247) [135](#_bookmark247)
   1. [Using an Atomikos transaction manager](#_bookmark248) [135](#_bookmark248)
   2. [Using a Bitronix transaction manager](#_bookmark249) [135](#_bookmark249)
   3. [Using a Narayana transaction manager](#_bookmark250) [136](#_bookmark250)
   4. [Using a Java EE managed transaction manager](#_bookmark251) [136](#_bookmark251)
   5. [Mixing XA and non-XA JMS connections](#_bookmark252) [136](#_bookmark252)
   6. [Supporting an alternative embedded transaction manager](#_bookmark253) [137](#_bookmark253)
5. [Hazelcast](#_bookmark254) [138](#_bookmark254)
6. [Spring Integration](#_bookmark255) [139](#_bookmark255)
7. [Spring Session](#_bookmark256) [140](#_bookmark256)
8. [Monitoring and management over JMX](#_bookmark257) [141](#_bookmark257)
9. [Testing](#_bookmark258) [142](#_bookmark258)
   1. [Test scope dependencies](#_bookmark259) [142](#_bookmark259)
   2. [Testing Spring applications](#_bookmark260) [142](#_bookmark260)
   3. [Testing Spring Boot applications](#_bookmark261) [143](#_bookmark261)

[Detecting test configuration](#_bookmark262) [143](#_bookmark262)

[Excluding test configuration](#_bookmark263) [144](#_bookmark263)

[Working with random ports](#_bookmark264) [144](#_bookmark264)

[Mocking and spying beans](#_bookmark265) [145](#_bookmark265)

[Auto-configured tests](#_bookmark266) [146](#_bookmark266)

[Auto-configured JSON tests](#_bookmark267) [146](#_bookmark267)

[Auto-configured Spring MVC tests](#_bookmark268) [147](#_bookmark268)

[Auto-configured Data JPA tests](#_bookmark269) [149](#_bookmark269)

[Auto-configured JDBC tests](#_bookmark270) [150](#_bookmark270)

[Auto-configured Data MongoDB tests](#_bookmark271) [150](#_bookmark271)

[Auto-configured REST clients](#_bookmark272) [151](#_bookmark272)

[Auto-configured Spring REST Docs tests](#_bookmark273) [152](#_bookmark273)

[Using Spock to test Spring Boot applications](#_bookmark274) [153](#_bookmark274)

* 1. [Test utilities](#_bookmark275) [153](#_bookmark275)

[ConfigFileApplicationContextInitializer](#_bookmark276) [153](#_bookmark276)

[EnvironmentTestUtils](#_bookmark277) [154](#_bookmark277)

[OutputCapture](#_bookmark278) [154](#_bookmark278)

[TestRestTemplate](#_bookmark279) [154](#_bookmark279)

1. [WebSockets](#_bookmark280) [156](#_bookmark280)
2. [Web Services](#_bookmark281) [157](#_bookmark281)
3. [Creating your own auto-configuration](#_bookmark282) [158](#_bookmark282)
   1. [Understanding auto-configured beans](#_bookmark283) [158](#_bookmark283)
   2. [Locating auto-configuration candidates](#_bookmark284) [158](#_bookmark284)
   3. [Condition annotations](#_bookmark285) [158](#_bookmark285)

[Class conditions](#_bookmark286) [159](#_bookmark286)

[Bean conditions](#_bookmark287) [159](#_bookmark287)

[Property conditions](#_bookmark288) [160](#_bookmark288)

[Resource conditions](#_bookmark289) [160](#_bookmark289)

[Web application conditions](#_bookmark290) [160](#_bookmark290)

[SpEL expression conditions](#_bookmark291) [160](#_bookmark291)

* 1. [Creating your own starter](#_bookmark292) [160](#_bookmark292)

[Naming](#_bookmark293) [160](#_bookmark293)

[Autoconfigure module](#_bookmark294) [161](#_bookmark294)

[Starter module](#_bookmark295) [161](#_bookmark295)

1. [What to read next](#_bookmark296) [162](#_bookmark296)
2. [Spring Boot Actuator: Production-ready features](#_bookmark297) [163](#_bookmark297)
3. [Enabling production-ready features](#_bookmark298) [164](#_bookmark298)
4. [Endpoints](#_bookmark299) [165](#_bookmark299)
   1. [Customizing endpoints](#_bookmark300) [166](#_bookmark300)
   2. [Hypermedia for actuator MVC endpoints](#_bookmark301) [167](#_bookmark301)
   3. [CORS support](#_bookmark302) [167](#_bookmark302)
   4. [Adding custom endpoints](#_bookmark303) [167](#_bookmark303)
   5. [Health information](#_bookmark304) [168](#_bookmark304)
   6. [Security with HealthIndicators](#_bookmark305) [168](#_bookmark305)

[Auto-configured HealthIndicators](#_bookmark306) [168](#_bookmark306)

[Writing custom HealthIndicators](#_bookmark307) [169](#_bookmark307)

* 1. [Application information](#_bookmark308) [170](#_bookmark308)

[Auto-configured InfoContributors](#_bookmark309) [170](#_bookmark309)

[Custom application info information](#_bookmark310) [170](#_bookmark310)

[Git commit information](#_bookmark311) [171](#_bookmark311)

[Build information](#_bookmark312) [171](#_bookmark312)

[Writing custom InfoContributors](#_bookmark313) [171](#_bookmark313)

1. [Monitoring and management over HTTP](#_bookmark314) [173](#_bookmark314)
   1. [Accessing sensitive endpoints](#_bookmark315) [173](#_bookmark315)
   2. [Customizing the management endpoint paths](#_bookmark316) [174](#_bookmark316)
   3. [Customizing the management server port](#_bookmark317) [174](#_bookmark317)
   4. [Configuring management-specific SSL](#_bookmark318) [175](#_bookmark318)
   5. [Customizing the management server address](#_bookmark319) [175](#_bookmark319)
   6. [Disabling HTTP endpoints](#_bookmark320) [175](#_bookmark320)
   7. [HTTP health endpoint format and access restrictions](#_bookmark321) [176](#_bookmark321)
2. [Monitoring and management over JMX](#_bookmark322) [178](#_bookmark322)
   1. [Customizing MBean names](#_bookmark323) [178](#_bookmark323)
   2. [Disabling JMX endpoints](#_bookmark324) [178](#_bookmark324)
   3. [Using Jolokia for JMX over HTTP](#_bookmark325) [178](#_bookmark325)

[Customizing Jolokia](#_bookmark326) [178](#_bookmark326)

[Disabling Jolokia](#_bookmark327) [178](#_bookmark327)

1. [Monitoring and management using a remote shell (deprecated)](#_bookmark328) [180](#_bookmark328)
   1. [Connecting to the remote shell](#_bookmark329) [180](#_bookmark329)

[Remote shell credentials](#_bookmark330) [181](#_bookmark330)

* 1. [Extending the remote shell](#_bookmark331) [181](#_bookmark331)

[Remote shell commands](#_bookmark332) [181](#_bookmark332)

[Remote shell plugins](#_bookmark333) [182](#_bookmark333)

1. [Loggers](#_bookmark334) [183](#_bookmark334)
   1. [Configure a Logger](#_bookmark335) [183](#_bookmark335)
2. [Metrics](#_bookmark336) [184](#_bookmark336)
   1. [System metrics](#_bookmark337) [184](#_bookmark337)
   2. [DataSource metrics](#_bookmark338) [185](#_bookmark338)
   3. [Cache metrics](#_bookmark339) [185](#_bookmark339)
   4. [Tomcat session metrics](#_bookmark340) [186](#_bookmark340)
   5. [Recording your own metrics](#_bookmark341) [186](#_bookmark341)
   6. [Adding your own public metrics](#_bookmark342) [186](#_bookmark342)
   7. [Special features with Java 8](#_bookmark343) [187](#_bookmark343)
   8. [Metric writers, exporters and aggregation](#_bookmark344) [187](#_bookmark344)

[Example: Export to Redis](#_bookmark345) [188](#_bookmark345)

[Example: Export to Open TSDB](#_bookmark346) [188](#_bookmark346)

[Example: Export to Statsd](#_bookmark347) [189](#_bookmark347)

[Example: Export to JMX](#_bookmark348) [189](#_bookmark348)

* 1. [Aggregating metrics from multiple sources](#_bookmark349) [190](#_bookmark349)
  2. [Dropwizard Metrics](#_bookmark350) [190](#_bookmark350)
  3. [Message channel integration](#_bookmark351) [191](#_bookmark351)

1. [Auditing](#_bookmark352) [192](#_bookmark352)
2. [Tracing](#_bookmark353) [193](#_bookmark353)
   1. [Custom tracing](#_bookmark354) [193](#_bookmark354)
3. [Process monitoring](#_bookmark355) [194](#_bookmark355)
   1. [Extend configuration](#_bookmark356) [194](#_bookmark356)
   2. [Programmatically](#_bookmark357) [194](#_bookmark357)
4. [Cloud Foundry support](#_bookmark358) [195](#_bookmark358)
   1. [Disabling extended Cloud Foundry actuator support](#_bookmark359) [195](#_bookmark359)
   2. [Cloud Foundry self signed certificates](#_bookmark360) [195](#_bookmark360)
   3. [Custom security configuration](#_bookmark361) [195](#_bookmark361)
5. [What to read next](#_bookmark362) [196](#_bookmark362)
6. [Deploying Spring Boot applications](#_bookmark363) [197](#_bookmark363)
7. [Deploying to the cloud](#_bookmark364) [198](#_bookmark364)
   1. [Cloud Foundry](#_bookmark365) [198](#_bookmark365)

[Binding to services](#_bookmark366) [199](#_bookmark366)

* 1. [Heroku](#_bookmark367) [199](#_bookmark367)
  2. [OpenShift](#_bookmark368) [201](#_bookmark368)
  3. [Amazon Web Services (AWS)](#_bookmark369) [201](#_bookmark369)

[AWS Elastic Beanstalk](#_bookmark370) [202](#_bookmark370)

[Using the Tomcat platform](#_bookmark371) [202](#_bookmark371)

[Using the Java SE platform](#_bookmark372) [202](#_bookmark372)

[Best practices](#_bookmark373) [202](#_bookmark373)

[Summary](#_bookmark374) [202](#_bookmark374)

* 1. [Boxfuse and Amazon Web Services](#_bookmark375) [202](#_bookmark375)
  2. [Google Cloud](#_bookmark376) [203](#_bookmark376)

1. [Installing Spring Boot applications](#_bookmark377) [205](#_bookmark377)
   1. [Supported operating systems](#_bookmark378) [205](#_bookmark378)
   2. [Unix/Linux services](#_bookmark379) [205](#_bookmark379)

[Installation as an init.d service (System V)](#_bookmark380) [205](#_bookmark380)

[Securing an init.d service](#_bookmark381) [206](#_bookmark381)

[Installation as a systemd service](#_bookmark382) [207](#_bookmark382)

[Customizing the startup script](#_bookmark383) [207](#_bookmark383)

[Customizing script when it’s written](#_bookmark384) [208](#_bookmark384)

[Customizing script when it runs](#_bookmark385) [209](#_bookmark385)

* 1. [Microsoft Windows services](#_bookmark387) [210](#_bookmark387)

1. [What to read next](#_bookmark388) [211](#_bookmark388)
2. [Spring Boot CLI](#_bookmark389) [212](#_bookmark389)
3. [Installing the CLI](#_bookmark390) [213](#_bookmark390)
4. [Using the CLI](#_bookmark391) [214](#_bookmark391)
   1. [Running applications using the CLI](#_bookmark392) [214](#_bookmark392)

[Deduced “grab” dependencies](#_bookmark393) [215](#_bookmark393)

[Deduced “grab” coordinates](#_bookmark394) [216](#_bookmark394)

[Default import statements](#_bookmark395) [216](#_bookmark395)

[Automatic main method](#_bookmark396) [216](#_bookmark396)

[Custom dependency management](#_bookmark397) [216](#_bookmark397)

* 1. [Testing your code](#_bookmark398) [217](#_bookmark398)
  2. [Applications with multiple source files](#_bookmark399) [217](#_bookmark399)
  3. [Packaging your application](#_bookmark400) [217](#_bookmark400)
  4. [Initialize a new project](#_bookmark401) [218](#_bookmark401)
  5. [Using the embedded shell](#_bookmark402) [218](#_bookmark402)
  6. [Adding extensions to the CLI](#_bookmark403) [219](#_bookmark403)

1. [Developing application with the Groovy beans DSL](#_bookmark404) [220](#_bookmark404)
2. [Configuring the CLI with settings.xml](#_bookmark405) [221](#_bookmark405)
3. [What to read next](#_bookmark406) [222](#_bookmark406)
4. [Build tool plugins](#_bookmark407) [223](#_bookmark407)
5. [Spring Boot Maven plugin](#_bookmark408) [224](#_bookmark408)
   1. [Including the plugin](#_bookmark409) [224](#_bookmark409)
   2. [Packaging executable jar and war files](#_bookmark410) [225](#_bookmark410)
6. [Spring Boot Gradle plugin](#_bookmark411) [226](#_bookmark411)
   1. [Including the plugin](#_bookmark412) [226](#_bookmark412)
   2. [Gradle dependency management](#_bookmark413) [226](#_bookmark413)
   3. [Packaging executable jar and war files](#_bookmark414) [226](#_bookmark414)
   4. [Running a project in-place](#_bookmark415) [227](#_bookmark415)
   5. [Spring Boot plugin configuration](#_bookmark416) [227](#_bookmark416)
   6. [Repackage configuration](#_bookmark417) [228](#_bookmark417)
   7. [Repackage with custom Gradle configuration](#_bookmark418) [229](#_bookmark418)

[Configuration options](#_bookmark419) [229](#_bookmark419)

[Available layouts](#_bookmark420) [230](#_bookmark420)

[Using a custom layout](#_bookmark421) [230](#_bookmark421)

* 1. [Understanding how the Gradle plugin works](#_bookmark422) [231](#_bookmark422)
  2. [Publishing artifacts to a Maven repository using Gradle](#_bookmark423) [231](#_bookmark423)

[Configuring Gradle to produce a pom that inherits dependency management](#_bookmark424) [231](#_bookmark424)

[Configuring Gradle to produce a pom that imports dependency management](#_bookmark425) [232](#_bookmark425)

1. [Spring Boot AntLib module](#_bookmark426) [233](#_bookmark426)
   1. [Spring Boot Ant tasks](#_bookmark427) [233](#_bookmark427)

[spring-boot:exejar](#_bookmark428) [233](#_bookmark428)

[Examples](#_bookmark429) [233](#_bookmark429)

* 1. [spring-boot:findmainclass](#_bookmark430) [234](#_bookmark430)

[Examples](#_bookmark431) [234](#_bookmark431)

1. [Supporting other build systems](#_bookmark432) [235](#_bookmark432)
   1. [Repackaging archives](#_bookmark433) [235](#_bookmark433)
   2. [Nested libraries](#_bookmark434) [235](#_bookmark434)
   3. [Finding a main class](#_bookmark435) [235](#_bookmark435)
   4. [Example repackage implementation](#_bookmark436) [235](#_bookmark436)
2. [What to read next](#_bookmark437) [236](#_bookmark437)
3. [‘How-to’ guides](#_bookmark438) [237](#_bookmark438)
4. [Spring Boot application](#_bookmark439) [238](#_bookmark439)
   1. [Create your own FailureAnalyzer](#_bookmark440) [238](#_bookmark440)
   2. [Troubleshoot auto-configuration](#_bookmark441) [238](#_bookmark441)
   3. [Customize the Environment or ApplicationContext before it starts](#_bookmark442) [239](#_bookmark442)
   4. [Build an ApplicationContext hierarchy (adding a parent or root context)](#_bookmark443) [240](#_bookmark443)
   5. [Create a non-web application](#_bookmark444) [240](#_bookmark444)
5. [Properties & configuration](#_bookmark445) [241](#_bookmark445)
   1. [Automatically expand properties at build time](#_bookmark446) [241](#_bookmark446)

[Automatic property expansion using Maven](#_bookmark447) [241](#_bookmark447)

[Automatic property expansion using Gradle](#_bookmark448) [242](#_bookmark448)

* 1. [Externalize the configuration of SpringApplication](#_bookmark449) [242](#_bookmark449)
  2. [Change the location of external properties of an application](#_bookmark450) [243](#_bookmark450)
  3. [Use ‘short’ command line arguments](#_bookmark451) [243](#_bookmark451)
  4. [Use YAML for external properties](#_bookmark452) [244](#_bookmark452)
  5. [Set the active Spring profiles](#_bookmark453) [244](#_bookmark453)
  6. [Change configuration depending on the environment](#_bookmark454) [244](#_bookmark454)
  7. [Discover built-in options for external properties](#_bookmark455) [245](#_bookmark455)

1. [Embedded servlet containers](#_bookmark456) [246](#_bookmark456)
   1. [Add a Servlet, Filter or Listener to an application](#_bookmark457) [246](#_bookmark457)

[Add a Servlet, Filter or Listener using a Spring bean](#_bookmark458) [246](#_bookmark458)

[Disable registration of a Servlet or Filter](#_bookmark459) [246](#_bookmark459)

[Add Servlets, Filters, and Listeners using classpath scanning](#_bookmark460) [246](#_bookmark460)

* 1. [Change the HTTP port](#_bookmark461) [247](#_bookmark461)
  2. [Use a random unassigned HTTP port](#_bookmark462) [247](#_bookmark462)
  3. [Discover the HTTP port at runtime](#_bookmark463) [247](#_bookmark463)
  4. [Configure SSL](#_bookmark464) [247](#_bookmark464)
  5. [Configure Access Logging](#_bookmark465) [248](#_bookmark465)
  6. [Use behind a front-end proxy server](#_bookmark466) [248](#_bookmark466)

[Customize Tomcat’s proxy configuration](#_bookmark467) [249](#_bookmark467)

* 1. [Configure Tomcat](#_bookmark468) [249](#_bookmark468)
  2. [Enable Multiple Connectors with Tomcat](#_bookmark469) [249](#_bookmark469)
  3. [Use Tomcat’s LegacyCookieProcessor](#_bookmark470) [250](#_bookmark470)
  4. [Use Jetty instead of Tomcat](#_bookmark471) [250](#_bookmark471)
  5. [Configure Jetty](#_bookmark472) [251](#_bookmark472)
  6. [Use Undertow instead of Tomcat](#_bookmark473) [251](#_bookmark473)
  7. [Configure Undertow](#_bookmark474) [252](#_bookmark474)
  8. [Enable Multiple Listeners with Undertow](#_bookmark475) [252](#_bookmark475)
  9. [Use Tomcat 7.x or 8.0](#_bookmark476) [252](#_bookmark476)

[Use Tomcat 7.x or 8.0 with Maven](#_bookmark477) [252](#_bookmark477)

[Use Tomcat 7.x or 8.0 with Gradle](#_bookmark478) [253](#_bookmark478)

* 1. [Use Jetty 9.2](#_bookmark479) [253](#_bookmark479)

[Use Jetty 9.2 with Maven](#_bookmark480) [253](#_bookmark480)

[Use Jetty 9.2 with Gradle](#_bookmark481) [253](#_bookmark481)

* 1. [Use Jetty 8](#_bookmark482) [253](#_bookmark482)

[Use Jetty 8 with Maven](#_bookmark483) [254](#_bookmark483)

[Use Jetty 8 with Gradle](#_bookmark484) [254](#_bookmark484)

* 1. [Create WebSocket endpoints using @ServerEndpoint](#_bookmark485) [254](#_bookmark485)
  2. [Enable HTTP response compression](#_bookmark486) [255](#_bookmark486)

1. [Spring MVC](#_bookmark487) [256](#_bookmark487)
   1. [Write a JSON REST service](#_bookmark488) [256](#_bookmark488)
   2. [Write an XML REST service](#_bookmark489) [256](#_bookmark489)
   3. [Customize the Jackson ObjectMapper](#_bookmark490) [256](#_bookmark490)
   4. [Customize the @ResponseBody rendering](#_bookmark491) [258](#_bookmark491)
   5. [Handling Multipart File Uploads](#_bookmark492) [258](#_bookmark492)
   6. [Switch off the Spring MVC DispatcherServlet](#_bookmark493) [258](#_bookmark493)
   7. [Switch off the Default MVC configuration](#_bookmark494) [259](#_bookmark494)
   8. [Customize ViewResolvers](#_bookmark495) [259](#_bookmark495)
   9. [Use Thymeleaf 3](#_bookmark496) [260](#_bookmark496)
2. [HTTP clients](#_bookmark497) [261](#_bookmark497)
   1. [Configure RestTemplate to use a proxy](#_bookmark498) [261](#_bookmark498)
3. [Logging](#_bookmark499) [262](#_bookmark499)
   1. [Configure Logback for logging](#_bookmark500) [262](#_bookmark500)

[Configure logback for file only output](#_bookmark501) [263](#_bookmark501)

* 1. [Configure Log4j for logging](#_bookmark502) [263](#_bookmark502)

[Use YAML or JSON to configure Log4j 2](#_bookmark503) [264](#_bookmark503)

1. [Data Access](#_bookmark504) [265](#_bookmark504)
   1. [Configure a custom DataSource](#_bookmark505) [265](#_bookmark505)
   2. [Configure Two DataSources](#_bookmark506) [267](#_bookmark506)
   3. [Use Spring Data repositories](#_bookmark507) [268](#_bookmark507)
   4. [Separate @Entity definitions from Spring configuration](#_bookmark508) [268](#_bookmark508)
   5. [Configure JPA properties](#_bookmark509) [268](#_bookmark509)
   6. [Configure Hibernate Naming Strategy](#_bookmark510) [269](#_bookmark510)
   7. [Use a custom EntityManagerFactory](#_bookmark511) [269](#_bookmark511)
   8. [Use Two EntityManagers](#_bookmark512) [269](#_bookmark512)
   9. [Use a traditional persistence.xml](#_bookmark513) [270](#_bookmark513)
   10. [Use Spring Data JPA and Mongo repositories](#_bookmark514) [270](#_bookmark514)
   11. [Expose Spring Data repositories as REST endpoint](#_bookmark515) [271](#_bookmark515)
   12. [Configure a component that is used by JPA](#_bookmark516) [271](#_bookmark516)
2. [Database initialization](#_bookmark517) [272](#_bookmark517)
   1. [Initialize a database using JPA](#_bookmark518) [272](#_bookmark518)
   2. [Initialize a database using Hibernate](#_bookmark519) [272](#_bookmark519)
   3. [Initialize a database](#_bookmark520) [272](#_bookmark520)
   4. [Initialize a Spring Batch database](#_bookmark521) [273](#_bookmark521)
   5. [Use a higher-level database migration tool](#_bookmark522) [273](#_bookmark522)

[Execute Flyway database migrations on startup](#_bookmark523) [273](#_bookmark523)

[Execute Liquibase database migrations on startup](#_bookmark524) [274](#_bookmark524)

1. [Messaging](#_bookmark525) [275](#_bookmark525)
   1. [Disable transacted JMS session](#_bookmark526) [275](#_bookmark526)
2. [Batch applications](#_bookmark527) [276](#_bookmark527)
   1. [Execute Spring Batch jobs on startup](#_bookmark528) [276](#_bookmark528)
3. [Actuator](#_bookmark529) [277](#_bookmark529)
   1. [Change the HTTP port or address of the actuator endpoints](#_bookmark530) [277](#_bookmark530)
   2. [Customize the ‘whitelabel’ error page](#_bookmark531) [277](#_bookmark531)
   3. [Actuator and Jersey](#_bookmark532) [277](#_bookmark532)
4. [Security](#_bookmark533) [278](#_bookmark533)
   1. [Switch off the Spring Boot security configuration](#_bookmark534) [278](#_bookmark534)
   2. [Change the AuthenticationManager and add user accounts](#_bookmark535) [278](#_bookmark535)
   3. [Enable HTTPS when running behind a proxy server](#_bookmark536) [278](#_bookmark536)
5. [Hot swapping](#_bookmark537) [280](#_bookmark537)
   1. [Reload static content](#_bookmark538) [280](#_bookmark538)
   2. [Reload templates without restarting the container](#_bookmark539) [280](#_bookmark539)

[Thymeleaf templates](#_bookmark540) [280](#_bookmark540)

[FreeMarker templates](#_bookmark541) [280](#_bookmark541)

[Groovy templates](#_bookmark542) [280](#_bookmark542)

* 1. [Fast application restarts](#_bookmark543) [280](#_bookmark543)
  2. [Reload Java classes without restarting the container](#_bookmark544) [281](#_bookmark544)

[Configuring Spring Loaded for use with Maven](#_bookmark545) [281](#_bookmark545)

[Configuring Spring Loaded for use with Gradle and IntelliJ IDEA](#_bookmark546) [281](#_bookmark546)

1. [Build](#_bookmark547) [283](#_bookmark547)
   1. [Generate build information](#_bookmark548) [283](#_bookmark548)
   2. [Generate git information](#_bookmark549) [283](#_bookmark549)
   3. [Customize dependency versions](#_bookmark550) [284](#_bookmark550)
   4. [Create an executable JAR with Maven](#_bookmark551) [284](#_bookmark551)
   5. [Use a Spring Boot application as a dependency](#_bookmark552) [285](#_bookmark552)
   6. [Extract specific libraries when an executable jar runs](#_bookmark553) [286](#_bookmark553)
   7. [Create a non-executable JAR with exclusions](#_bookmark554) [286](#_bookmark554)
   8. [Remote debug a Spring Boot application started with Maven](#_bookmark555) [287](#_bookmark555)
   9. [Remote debug a Spring Boot application started with Gradle](#_bookmark556) [287](#_bookmark556)
   10. [Build an executable archive from Ant without using spring-boot-antlib](#_bookmark557) [288](#_bookmark557)
   11. [How to use Java 6](#_bookmark558) [289](#_bookmark558)

[Embedded servlet container compatibility](#_bookmark559) [289](#_bookmark559)

[Jackson](#_bookmark560) [289](#_bookmark560)

[JTA API compatibility](#_bookmark561) [289](#_bookmark561)

1. [Traditional deployment](#_bookmark562) [290](#_bookmark562)
   1. [Create a deployable war file](#_bookmark563) [290](#_bookmark563)
   2. [Create a deployable war file for older servlet containers](#_bookmark564) [291](#_bookmark564)
   3. [Convert an existing application to Spring Boot](#_bookmark565) [291](#_bookmark565)
   4. [Deploying a WAR to WebLogic](#_bookmark566) [293](#_bookmark566)
   5. [Deploying a WAR in an Old (Servlet 2.5) Container](#_bookmark567) [293](#_bookmark567)
2. [Appendices](#_bookmark568) [295](#_bookmark568)
3. [Common application properties](#_bookmark569) [296](#_bookmark569)
4. [Configuration meta-data](#_bookmark571) [318](#_bookmark571)
   1. [Meta-data format](#_bookmark572) [318](#_bookmark572)

[Group Attributes](#_bookmark573) [319](#_bookmark573)

[Property Attributes](#_bookmark574) [320](#_bookmark574)

[Hint Attributes](#_bookmark575) [322](#_bookmark575)

[Repeated meta-data items](#_bookmark576) [322](#_bookmark576)

* 1. [Providing manual hints](#_bookmark577) [323](#_bookmark577)

[Value hint](#_bookmark578) [323](#_bookmark578)

[Value provider](#_bookmark579) [323](#_bookmark579)

[Any](#_bookmark580) [324](#_bookmark580)

[Class reference](#_bookmark581) [325](#_bookmark581)

[Handle As](#_bookmark582) [325](#_bookmark582)

[Logger name](#_bookmark583) [326](#_bookmark583)

[Spring bean reference](#_bookmark584) [327](#_bookmark584)

[Spring profile name](#_bookmark585) [328](#_bookmark585)

* 1. [Generating your own meta-data using the annotation processor](#_bookmark586) [328](#_bookmark586)

[Nested properties](#_bookmark587) [329](#_bookmark587)

[Adding additional meta-data](#_bookmark588) [329](#_bookmark588)

1. [Auto-configuration classes](#_bookmark589) [331](#_bookmark589)
   1. [From the “spring-boot-autoconfigure” module](#_bookmark590) [331](#_bookmark590)
   2. [From the “spring-boot-actuator” module](#_bookmark591) [334](#_bookmark591)
2. [Test auto-configuration annotations](#_bookmark592) [336](#_bookmark592)
3. [The executable jar format](#_bookmark593) [338](#_bookmark593)
   1. [Nested JARs](#_bookmark594) [338](#_bookmark594)

[The executable jar file structure](#_bookmark595) [338](#_bookmark595)

[The executable war file structure](#_bookmark596) [338](#_bookmark596)

* 1. [Spring Boot’s “JarFile” class](#_bookmark597) [339](#_bookmark597)

[Compatibility with the standard Java “JarFile”](#_bookmark598) [339](#_bookmark598)

* 1. [Launching executable jars](#_bookmark599) [339](#_bookmark599)

[Launcher manifest](#_bookmark600) [340](#_bookmark600)

[Exploded archives](#_bookmark601) [340](#_bookmark601)

* 1. [PropertiesLauncher Features](#_bookmark602) [340](#_bookmark602)
  2. [Executable jar restrictions](#_bookmark603) [342](#_bookmark603)

[Zip entry compression](#_bookmark604) [342](#_bookmark604)

[System ClassLoader](#_bookmark605) [342](#_bookmark605)

* 1. [Alternative single jar solutions](#_bookmark606) [342](#_bookmark606)

1. [Dependency versions](#_bookmark607) [343](#_bookmark607)

**第一部分. Spring Boot 文档**

本章节简要介绍Spring Boot参考文档。将其视为该文档的其余部分的地图。可以以线性方式阅读本参考文档，或者如果你不感兴趣，可以跳过本分内容。

# 关于文档

Spring Boot参考指南可作为html,pdf和epub文档。最新的副本可在官网获得。

本文件以副本可供自己使用，此类副本的额任何费用的情况下可以分发发给他人，，并进一步规定，不论是以书面形式还是以电子邮件方式分发，每个副本均包含本版权声明。

# 获取帮助

有关Spring Boot的问题，我们愿意提供帮助！

* [stackoverflow.com](http://stackoverflow.com/)
* [github.com/spring-projects/spring-boot/issues](https://github.com/spring-projects/spring-boot/issues).

# 第一步

如果你刚开始使用Spring Boot，或者通常是Spring，这是一个开始使用Spring Boot 的地方。

* **从头开始:** [概述](#_bookmark9) | [要求](#_bookmark10) | [安装](#_bookmark12)
* **教程:** [Part 1](#_bookmark24) | [Part 2](#_bookmark27)
* **运行示例:** [Part 1](#_bookmark31) | [Part 2](#_bookmark32)

# 使用 Spring Boot

准备好开始使用Spring Boot了吗？

Ready to actually start using Spring Boot? [我们默认已经准备好](#_bookmark34).

* **构建系统:** [Maven](#_bookmark37) | [Gradle](#_bookmark42) | [Ant](#_bookmark43) | [Starters](#_bookmark44)
* **最佳做法:** [Code Structure](#_bookmark50) | [@Configuration](#_bookmark53) | [@EnableAutoConfiguration](#_bookmark56) | [Beans and Dependency Injection](#_bookmark59)
* **运行你的代码** [IDE](#_bookmark62) | [Packaged](#_bookmark63) | [Maven](#_bookmark64) | [Gradle](#_bookmark65)
* **打包你的应用程序:** [Production jars](#_bookmark83)
* **启动Spring Boot:** [Using the CLI](#_bookmark389)

# 学习 Spring Boot 特性

需要更多关于Spring Boot的核心特性？这里都有。

* **核心特性:** [SpringApplication](#_bookmark86) | [External Configuration](#_bookmark97) | [Profiles](#_bookmark115) | [Logging](#_bookmark119)
* **Web 应用程序:** [MVC](#_bookmark130) | [Embedded Containers](#_bookmark146)
* **数据处理:** [SQL](#_bookmark165) | [NO-SQL](#_bookmark183)
* **消息:** [Overview](#_bookmark228) | [JMS](#_bookmark229)
* **测试:** [Overview](#_bookmark258) | [Boot Applications](#_bookmark261) | [Utils](#_bookmark275)
* **扩展:** [Auto-configuration](#_bookmark282) | [@Conditions](#_bookmark285)

# 发布到生产环境

当你准备将Spring Boot应用程序部署到生产环境，[技巧](#_bookmark297) 你可能会喜欢。

* **管理端点:** [Overview](#_bookmark299) | [Customization](#_bookmark300)
* **连接选项:** [HTTP](#_bookmark314) | [JMX](#_bookmark322) | [SSH](#_bookmark328)
* **监控:** [Metrics](#_bookmark336) | [Auditing](#_bookmark352) | [Tracing](#_bookmark353) | [Process](#_bookmark355)

# 高级主题

最后，我们为更高级的用户提供了一些主题。

* **部署Spring Boot应用程序:** [云部署](#_bookmark364) | [OS服务](#_bookmark379)
* **构建工具插件:** [Maven](#_bookmark408) | [Gradle](#_bookmark411)
* **附录:** [Application Properties](#_bookmark569) | [Auto-configuration classes](#_bookmark589) | [Executable Jars](#_bookmark593)

**第二部分. 入门**

如果你刚开始使用Spring Boot，或者一般来说就是Spring，那么这部分是你所需要了解的！在这里我们回答基本“what？”，“how?”,”why?”的问题。你会发现一个优美的介绍，以及安装说明。然后，我们将构建我们的第一个Spring Boot应用程序，并讨论一些核心原则。

# Spring Boot 介绍

Spring Boot 是你可以轻松创建单机的生产级别的基于Spring的可运行的应用程序。我们队Spring平台和第三方库有自己的一些做法，所以你可以从最小的功能开始。大多数Spring Boot 应用程序值需要很少的Spring配置。

你可以使用Spring Boot创建可以用java –jar或更传统的war部署来启动的应用程序。我们还提供了一个运行“spring scripts”的命令行工具。

我们的主要目标是：

* 为所有Spring开发提供一个更快，更广泛的入门体验。
* 开箱即用，但随着需求开始偏离默认值，快速开始。
* 提供大量项目(如嵌入式服务器，安全性，指标，运行状况检查，外部配置)通用的一些列非功能性功能。
* 绝对不生成代码，不需要XML配置

# 系统要求

默认情况下，Spring Boot1.5.8.RELEASE需要Java 7 和 Spring Framework 4.3.12.RELEASE或更高版本。你可以使用带有Java 6的Spring Boot以及其他一些配置。有关更多详细信息请参见84.11节如何使用Java 6 。Maven(3.2+)和Gradle 2(2.9或更高版本)和3提供了明确的构建支持。

**Tip**

虽然可以使用Java 6 或 7 的Spring Boot，但是我们通常推荐使用Java 8 。

## Servlet 容器

开箱即用支持一下嵌入式servlet容器:

|  |  |
| --- | --- |
| **Name** | **Servlet Version Java Version** |
| Tomcat 8  Tomcat 7 | 3.1 Java 7+  3.0 Java 6+ |
| Jetty 9.3 | 3.1 Java 8+ |
| Jetty 9.2 | 3.1 Java 7+ |
| Jetty 8 | 3.0 Java 6+ |
| Undertow 1.3 | 3.1 Java 7+ |

你也可以将Spring Boot应用程序部署到任何与Servlet 3.0+兼容的容器。

# 安装 Spring Boot

Spring Boot可以与经典Java开发工具一期使用，也可以作为命令行工具安装。无论如何，你将需要Java SDK V1.6 或更高版本。在空时之前，你应该检查你当前的Java安装:

$ java -version

如果你对Java开发不熟悉，或者你知识向尝试Spring Boot, 则可能需要先尝试Spring Boot CLI，否则请阅读[Spring Boot CLI](#_bookmark16) 安装说明。

**Tip**

尽管Spring Boot与Java 6兼容，但如果可能的话，应该考虑使用最新版本的Java

## Java开发人员的安装说明

你可以像使用任何标准Java库一样使用Spring Boot。只需要在你的类路径中包含相应的spring-boot.-\*.jar文件即可。Spring Boot不需要任何特殊的工具集成，所以可以使用任何IDE或文本编辑器;Spring Boot应用程序没有什么特别之处，所以你可以像运行其他Java程序一样运行和调试。

尽管你可以复制Spring Boot jars，但我们通常建议你使用支持依赖管理的构建工具(如Maven或Gradle)。

### Maven 安装

Spring Boot与Apache Maven3.2或更高版本兼容。如果你还没有安装maven，你可以按照[maven.apache.org](http://maven.apache.org/)上的说进行操作。

**Tip**

在许多操作系统上，maven可以通过包管理器来安装。如果你是OSX 用户，请尝试安装maven。ubuntu用于而已运行sudo apt-get install maven。

Spring Boot 依赖使用org.springframework.boot groupId。通常，你的maven POM文件将从spring-boot-starter-parent项目继承，并向一个或多个starter声明依赖关系。Spring Boot还提供一个可选的maven插件来创建可执行的jar文件。

这是一个典型的 pom.xml 文件:

<?xml version="1.0" encoding="UTF-8"?>

**<project xmlns**=[**"http://maven.apache.org/POM/4.0.0"**](http://maven.apache.org/POM/4.0.0) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://maven.apache.org/POM/4.0.0**](http://maven.apache.org/POM/4.0.0)[**http://maven.apache.org/xsd/maven-4.0.0.xsd"**](http://maven.apache.org/xsd/maven-4.0.0.xsd)**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<groupId>**com.example**</groupId>**

**<artifactId>**myproject**</artifactId>**

**<version>**0.0.1-SNAPSHOT**</version>**

*<!-- Inherit defaults from Spring Boot -->*

**<parent>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**</parent>**

*<!-- Add typical dependencies for a web application -->*

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**</dependencies>**

*<!-- Package as an executable jar -->*

**<build>**

**<plugins>**

**<plugin>**

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

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

**</plugin>**

**</plugins>**

**</build>**

**</project>**

**Tip**

spring-boot-starter-parent 是使用Spring Boot的好方法，但他可能并不适合所有的情况。有时呢可能需要从不同的父POM继承，或者你可能不喜欢我们的默认设置。有关使用import scope 的替代解决方案，请参考[POM的Spring Boot章节](#_bookmark39)

## 安装 Spring Boot CLI

Spring Boot CLI是一个命令行工具，可以使用它快速创建Spring原型。 它允许你运行Groovy脚本，这意味着你有一个数据的类Java语法，没有太多的样板代码。

你不需要使用CLI来使用Spring Boot，但它绝对是让Spring应用程序实现最快速快捷的方式。

### 手动安装

你可以从Spring软件存储库下载Spring CLI发行版：

* [spring-boot-cli-1.5.8.RELEASE-bin.zip](http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.5.8.RELEASE/spring-boot-cli-1.5.8.RELEASE-bin.zip)
* [spring-boot-cli-1.5.8.RELEASE-bin.tar.gz](http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.5.8.RELEASE/spring-boot-cli-1.5.8.RELEASE-bin.tar.gz)

最新版本 [snapshot distributions](http://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/) 也是可以使用的。

下载之后，请按照解压缩归档中的[INSTALL.txt](https://raw.github.com/spring-projects/spring-boot/v1.5.8.RELEASE/spring-boot-cli/src/main/content/INSTALL.txt) 说明进行操作。总结：在.zip文件的bin目录下有一个spring脚本(用于Windows的spring.bat)，或可以使用带有.jar文件的java –jar(该脚本可以帮助你确保类路径设置正确)。

### 使用 SDKMAN安装!

SDKMAN!(软件开发工具包管理器)可用于管理各种二进制SDK的多个版本，包括Groovy和Spring Boot CLI。获取[SDKMAN](http://sdkman.io/)! 从sdkman.io安装Spring Boot。

$ sdk install springboot

$ spring --version

Spring Boot v1.5.8.RELEASE

如果你正在开发CLI的功能并希望轻松访问你刚刚构建的版本，请按照这些额外说明进行操作。

$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-1.5.8.RELEASE- bin/spring-1.5.8.RELEASE/

$ sdk default springboot dev

$ spring --version

Spring CLI v1.5.8.RELEASE

这将安装一个名为dev实例的spring的本地实例。它指向你的目标构建位置，所以每次重建Spring Boot时，spring都是最新的。

你可以如下这样做::

$ sdk ls springboot

================================================================================

Available Springboot Versions

================================================================================

> + dev

\* 1.5.8.RELEASE

================================================================================

+ - local version

\* - installed

> - currently in use

================================================================================

### OSX Homebrew 安装

如果你在Mac上并使用Homebrew，则只需要安装Spring Boot CLI即可：

$ brew tap pivotal/tap

$ brew install springboot

Homebrew 将会将spring安装到 /usr/local/bin.

**Note**

如果你没有看到该公式，那么你的brew的安装可能会过期。值需要执行brew更新并重启。

### MacPorts 安装

如果你在Mac上并使用[MacPorts](http://www.macports.org/)，则只需安装Spring Boot CLI即可：

$ sudo port install spring-boot-cli

### 命令行完成

Spring Boot CLI提供了为[BASH](http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29) 和[zsh](http://en.wikipedia.org/wiki/Zsh) shells命令完成脚本。你可以在任何shell中获取脚本(也称为spring)的源代码，或者将其放入个人或系统范围的bash完成初始化。在Debian系统上，系统范围的焦恩在/shell-completion/bash中，当一个新的shell启动时，该目录中的所有脚本都会被执行。要手动运行脚本，例如:如果你已经安装使用SDKMAN!

$ . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring

$ spring <HIT TAB HERE>

grab help jar run test version

**Note**

如果你使用Homebrew或MacPorts安装Spring Boot CLI，则命令行完成脚本会自动在您的shell中注册。

### 快速启动 Spring CLI 示例

这是一个非常建档的web应用程序，你可以使用它来测试你的安装。创建一个名为app.groovy的文件。

@RestController

**class** ThisWillActuallyRun {

@RequestMapping("/") String home() {

***"Hello World!"***

}

}

然后简单地从一个shell运行它：

$ spring run app.groovy

**Note**

首次运行应用程序需要一段时间，因为依赖关系将会被下载。后续运行将会更快。

在你喜欢的浏览器中打开localhost:8080 ，你应该看到下面的输出。

Hello World!

## 从早期版本的Spring Boot升级

如果你是从早期版本的Spring Boot进行升级，请查看[project wiki](https://github.com/spring-projects/spring-boot/wiki)发行说明。你会发现升级说明以及每个版本的新功能和指的注意功能列表。

要升级现有的CLI安装，请使用响应的package manager命令(如brew升级)，或者如果你手动安装了CLI，请按照标准说明(standard instructions)记住更新PATH环境变量以及删除所有旧的引用。

# 开发你的第一个Spring Boot 应用程序

我们用Java开发一个简单的“Hello World！”web应用程序，重点介绍Spring Boot的一些主要特性。我们将使用Maven来构建这个项目，因为大多说IDE都支持。

**Tip**

spring.io网站包含许多使用Spring Boot 的入门指南。如果你想解决一个待定的问题，现在哪里查看。

你可以通过转到[start.spring.io](https://start.spring.io/) 并从依赖关系搜索器中选择web starter来快速执行以下步骤。这将自动商城一个新的项目结构，以便你可以立即开始编码([start coding right away](#_bookmark27))。检查文档以获取更多详细信息([documentation for more details](https://github.com/spring-io/initializr))。

在开始之前，请打开终端，检查是否安装了Java和Maven的有效版本。

$ java -version

java version "1.7.0\_51"

Java(TM) SE Runtime Environment (build 1.7.0\_51-b13)

Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)

$ mvn -v

Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T13:58:10-07:00) Maven home: /Users/user/tools/apache-maven-3.1.1

Java version: 1.7.0\_51, vendor: Oracle Corporation

**Note**

此示例需要在自己的文件中创建。后续的说明嘉定你已经创建了一个合适的文件夹，并且它是你的当前目录。

## 创建 POM

我们需要从创建一个maven pom.xml文件开始。pom.xml是用来构建项目的配方。打开你喜欢的文本编辑器并添加一下内容。

<?xml version="1.0" encoding="UTF-8"?>

**<project xmlns**=[**"http://maven.apache.org/POM/4.0.0"**](http://maven.apache.org/POM/4.0.0) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://maven.apache.org/POM/4.0.0**](http://maven.apache.org/POM/4.0.0)[**http://maven.apache.org/xsd/maven-4.0.0.xsd"**](http://maven.apache.org/xsd/maven-4.0.0.xsd)**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<groupId>**com.example**</groupId>**

**<artifactId>**myproject**</artifactId>**

**<version>**0.0.1-SNAPSHOT**</version>**

**<parent>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**</parent>**

*<!-- Additional lines to be added here... -->*

**</project>**

这应该给你一个工作的构建，你可以通过运行mvn(你可以忽略jar僵尸空的 – 没有内容被标记为包含警告)来测试它。

**Note**

此时，你可以将项目导入IDE(大多数现代Java IDE包含多maven的内置支持)。为了简单起见，我们将继续在这个例子中使用纯文本编辑器。

## 添加 classpath 依赖

Spring Boot 提供了许多“starters”，可以方便地将jar添加到你的classpath中。我们的示例应用程序已经在pom的父节点中使用了spring-boot-starter-parent。spring-boot-starter-parent是一个特别的启动器，提供了有用的maven默认值。它还提供了一个依赖管理部分，以便你可以省略版本标记获得“blessed”的依赖关系。

其他入门者知识提供开发特定类型的应用程序时可能需要的依赖关系。由于我们正在开发一个web应用程序，我们将添加一个spring-boot-start-web依赖项 – 但在此之前，让我们看看我们目前有什么。

.mvn 依赖项：tree命令打印项目依赖项的属性结构。

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**</dependencies>**

$ mvn dependency:tree

[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT

你可以看到spring-boot-starter-parent本身不提供依赖关系。 让我们编辑我们的pom.xml，并在父节的下方添加spring-boot-starter-web依赖项：

如果再次运行mvn dependency：tree，则会看到现在还有许多附加依赖项，包括Tomcat Web服务器和Spring Boot本身。.

## 写 code

为了完成我们的应用程序，我们需要创建一个Java文件。 Maven默认会从src / main / java编译源代码，所以你需要创建这个文件夹结构，然后添加一个名为src / main / java / Example.java的文件：

**import** org.springframework.boot.\*;

**import** org.springframework.boot.autoconfigure.\*;

**import** org.springframework.stereotype.\*;

**import** org.springframework.web.bind.annotation.\*;

@RestController @EnableAutoConfiguration **public class** Example {

@RequestMapping("/") String home() {

**return *"Hello World!"***;

}

**public static void** main(String[] args) **throws** Exception { SpringApplication.run(Example.**class**, args);

}

}

虽然这里没有太多的代码，但还是有很多。 我们来看看重要的部分。

### @RestController 和 @RequestMapping 注解

我们的Example类的第一个注解是@RestController。 这被称为*stereotype注解*。 它为阅读代码提供了线索，对于Spring来说，这个类扮演着特定的角色。 在这种情况下，我们的类是一个web @Controller，所以Spring在处理传入的Web请求时会考虑它。

@RequestMapping注释提供了“路由”信息。 它告诉Spring，任何具有路径“/”的HTTP请求都应该映射到home方法。 @RestController注释告诉Spring将结果字符串直接返回给调用者。

**Tip**

@RestController和@RequestMapping注解是Spring MVC注释（它们不是Spring Boot特有的）。 有关更多详细信息，请参阅[MVC section](https://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#mvc)参考资料中的MVC部分。

### @EnableAutoConfiguration 注解

第二个类级注释是@EnableAutoConfiguration。 这个注解告诉Spring Boot根据你添加的jar依赖来“猜测”你将如何配置Spring。 由于spring-boot-starter-web添加了Tomcat和Spring MVC，所以自动配置会假定你正在开发一个Web应用程序并相应地设置Spring。

|  |
| --- |
| **Starters and Auto-Configuration**  自动配置旨在与“启动器”配合使用，但这两个概念并不直接相关。 您可以自由选择和选择初学者之外的jar依赖项，并且Spring Boot将尽其所能地自动配置您的应用程序。 |

### “main” 方法

我们的应用程序的最后一部分是主要的方法。 这只是一个遵循Java约定的应用程序入口点的标准方法。 我们的主要方法通过调用run来委托Spring Boot的SpringApplication类。 SpringApplication将引导我们的应用程序，从Spring开始，它将启动自动配置的Tomcat Web服务器。 我们需要将Example.class作为参数传递给run方法，以告知SpringApplication是Spring的主要组件。 args数组也被传递以暴露任何命令行参数。.

## 运行这个示例

在这一点上我们的应用程序应该工作 由于我们使用了spring-boot-starter-parent POM，所以我们有一个有用的运行目标，我们可以使用它来启动应用程序。 键入mvn spring-boot：run 运行以启动应用程序：

\_ \_

\_

.

$ mvn spring-boot:run

/\\ / '\_ \_ \_(\_)\_ \_ \ \ \ \ ( ( )\ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ )| |\_)| | | | | || (\_| | ) ) ) ) ' | | . |\_| |\_|\_| |\_\ , | / / / /

=========|\_|==============| /=/\_/\_/\_/

:: Spring Boot :: (v1.5.8.RELEASE)

....... . . .

....... . . . (log output here)

....... . . .

........ Started Example in 2.222 seconds (JVM running for 6.514)

如果你打开一个web浏览器到localhost：8080，你应该看到如下输出：

Hello World!

要正常退出应用程序点击ctrl-c。

## 创建一个可执行的 jar

让我们通过创建一个完全独立的可执行jar文件来完成我们的例子，我们可以在生产环境中运行它。 可执行jar（有时也称为“fat jars”）是包含您编译的类以及您的代码需要运行的所有jar依赖项的归档文件。

|  |
| --- |
| **Executable jars and Java**  Java不提供任何标准的方法来加载嵌套的jar文件（即jar文件本身包含在jar中）。 如果您正在分发自包含的应用程序，这可能会有问题。  为了解决这个问题，许多开发者使用“uber”jar。 “uber”jar只是将所有jar里的所有类打包成一个单一的档案。 这种方法的问题在于，很难在应用程序中看到实际使用的库。 如果在多个罐子中使用相同的文件名（但具有不同的内容），则也可能是有问题的。  Spring Boot采用了不同的方法，可以直接嵌入jar。 |

要创建一个可执行的jar文件，我们需要将spring-boot-maven-plugin添加到我们的pom.xml。 在依赖关系部分下面插入以下几行：

**<build>**

**<plugins>**

**<plugin>**

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

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

**</plugin>**

**</plugins>**

**</build>**

**Note**

spring-boot-starter-parent POM包含<executions>配置来绑定重新打包目标。 如果您不使用父POM，则需要自行声明此配置。 有关详细信息，请参阅插件文档([plugin documentation](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/usage.html) )。

保存你的pom.xml并从命令行运行mvn package：

$ mvn package

[INFO] Scanning for projects... [INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building myproject 0.0.1-SNAPSHOT

[INFO] ------------------------------------------------------------------------ [INFO] .... ..

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---

[INFO] Building jar: /Users/developer/example/spring-boot-example/target/myproject-0.0.1-SNAPSHOT.jar [INFO]

[INFO] --- spring-boot-maven-plugin:1.5.8.RELEASE:repackage (default) @ myproject --- [INFO] ------------------------------------------------------------------------ [INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

如果您查看目标目录，则应该看到myproject-0.0.1-SNAPSHOT.jar。 该文件大小应该在10 MB左右。 如果你想偷看里面，你可以使用jar tvf：

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

您还应该在目标目录中看到一个名为myproject-0.0.1-SNAPSHOT.jar.original的小得多的文件。 这是Maven在被Spring Boot重新包装之前创建的原始jar文件。

使用 java –jar 命 令 运 行 jj a r 文 件 。

/\\ / '\_ \_ \_(\_)\_ \_ \ \ \ \ ( ( )\ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ )| |\_)| | | | | || (\_| | ) ) ) ) ' | | . |\_| |\_|\_| |\_\ , | / / / /

=========|\_|==============| /=/\_/\_/\_/

:: Spring Boot :: (v1.5.8.RELEASE)

....... . . .

....... . . . (log output here)

....... . . .

........ Started Example in 2.536 seconds (JVM running for 2.864)

\_ \_

\_

.

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

像以前一样，要优雅地退出应用程序点击ctrl-c。

# 接下来读什么

希望本节为您提供了一些Spring Boot基础知识，并帮助您编写自己的应用程序。 如果您是面向任务的开发人员，则可能需要跳过[spring.io](https://spring.io/)，查看一些入门指南(getting started)，以解决具体的“如何使用Spring”问题; 我们也有Spring Boot特定( [*How-to*](#_bookmark438) )的操作指南文档。

Spring Boot版本([Spring Boot repository](http://github.com/spring-projects/spring-boot) )也有一些你可以运行的样本([bunch of samples](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples) )。 样本独立于代码的其余部分（也就是说，您不需要构建其余的代码来运行或使用样本）。

另外，下一个逻辑步骤是阅读[*Part III, “Using Spring Boot”*](#_bookmark34)。 如果你真的不耐烦，也可以跳过来阅读Spring Boot的特([*Spring Boot features*](#_bookmark85))。

**第三部分. 使用 Spring Boot**

本节将更详细地介绍如何使用Spring Boot。 它涵盖了构建系统，自动配置以及如何运行应用程序等主题。 我们还介绍了一些Spring Boot的最佳实践。 尽管Spring Boot没有什么特别的地方（它只是你可以使用的另一个库），但是有一些建议，如果遵循这些建议，将使开发过程变得更容易一些。

如果您刚刚开始使用Spring Boot，则可能需要先阅读[*Getting Started*](#_bookmark8) ，然后再深入本节。

# 构建 systems

强烈建议您选择支持依赖管理的构建系统([*dependency management*](#_bookmark36),)，并且可以使用发布到“Maven Central”存储库的工件。 我们建议您选择Maven或Gradle。 Spring Boot可以与其他构建系统（例如Ant）一起工作，但是它们不会得到特别好的支持。

## Dependency 管理

Spring Boot的每个发行版都提供了一个支持的依赖列表。 在实践中，您不需要为构建配置中的任何这些依赖项提供一个版本，因为Spring Boot正在为您进行管理。 当您升级Spring Boot本身时，这些依赖关系也将以一致的方式升级。

**Note**

如果您觉得有必要，您仍然可以指定一个版本并覆盖Spring Boot的建议。

策划的列表包含您可以使用Spring Boot的所有spring模块以及第三方库的精炼列表。 该列表可用作标准物料清单（[Bills of Materials (spring-boot-](#_bookmark39) [dependencies)](#_bookmark39) ，并且还提供[Maven](#_bookmark38) 和[Gradle](#_bookmark413) 的其他专用支持。

**Warning**

Spring Boot的每个版本都与Spring框架的基础版本相关联，所以我们

强烈建议您不要自行指定版本。

## Maven

Maven用户可以继承spring-boot-starter-parent项目以获得合理的默认值。 父项目提供以下功能：

* Java 1.6 作为默认的编译器级别
* UTF-8 源代码格式
* [Dependency Management section](#_bookmark36),，允许您省略继承自Spring-Boot-Dependency POM的通用依赖项的<version>标记。
* 智能 [resource filtering](https://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html).
* 智能 plugin configuration ([exec plugin](http://www.mojohaus.org/exec-maven-plugin/), [surefire](http://maven.apache.org/surefire/maven-surefire-plugin/), [Git commit ID](https://github.com/ktoso/maven-git-commit-id-plugin), [shade](http://maven.apache.org/plugins/maven-shade-plugin/)).
* 智能 resource filtering for application.properties and application.yml 包括 profile-specific files (e.g. application-foo.properties and application-foo.yml)

最后一点：由于默认配置文件接受Spring样式占位符（$ {...}），Maven过滤被改为使用@ .. @占位符（您可以用Maven属性resource.delimiter覆盖）。

### 继承 starter parent

要将项目配置为从spring-boot-starter-parent继承，只需设置父项：

*<!-- Inherit defaults from Spring Boot -->*

**<parent>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**</parent>**

**Note**

您应该只需要在此依赖项上指定Spring Boot版本号。 如果您导入更多的启动器，则可以安全地省略版本号。

通过该设置，您还可以通过在自己的项目中重写属性来覆盖各个依赖项。 例如，要升级到另一个Spring Data发行版，您需要将以下内容添加到您的pom.xml中

**Tip**

**<properties>**

**<spring-data-releasetrain.version>**Fowler-SR2**</spring-data-releasetrain.version>**

**</properties>**

检查 [spring-boot-dependencies pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-dependencies/pom.xml) 以获取支持的列表。

### 使用没有父 POM 的 Spring Boot

不是每个人都喜欢从spring-boot-starter-parent POM继承。 你可能有你自己的企业标准的父母，你需要使用，或者你可能只是喜欢显式声明所有的Maven配置。

如果您不想使用spring-boot-starter-parent，那么仍然可以通过使用scope = import dependency来保留依赖关系管理（而不是插件管理）的。

**<dependencyManagement>**

**<dependencies>**

**<dependency>**

*<!-- Import dependency management from Spring Boot -->*

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

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

**<version>**1.5.8.RELEASE**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>**

**</dependency>**

**</dependencies>**

**</dependencyManagement>**

如上所述，该设置不允许您使用属性覆盖单个依赖项。 为了达到同样的结果，你需要在spring-boot-dependencies条目之前在项目的dependencyManagement中添加一个条目。 例如，要升级到另一个Spring Data发行版，您需要将以下内容添加到您的pom.xml中。

**<dependencyManagement>**

**<dependencies>**

*<!-- Override Spring Data release train provided by Spring Boot -->*

**<dependency>**

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

**<artifactId>**spring-data-releasetrain**</artifactId>**

**<version>**Fowler-SR2**</version>**

**<scope>**import**</scope>**

**<type>**pom**</type>**

**</dependency>**

**<dependency>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>**

**</dependency>**

**</dependencies>**

**</dependencyManagement>**

**Note**

在上面的例子中，我们指定了一个BOM，但是任何依赖类型都可以被覆盖。

### 更改 Java 版本

spring-boot-starter-parent选择相当保守的Java兼容性。 如果您想遵循我们的建议并使用较新的Java版本，则可以添加一个java.version属性：

**<properties>**

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

**</properties>**

### 使用 Spring Boot Maven 插件

Spring Boot包含一个Maven插件，可以将项目打包为可执行的jar文件。 如果要使用它，请将插件添加到<plugins>部分

**<build>**

**<plugins>**

**<plugin>**

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

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

**</plugin>**

**</plugins>**

**</build>**

**Note**

如果你使用Spring Boot starter parent pom POM，你只需要添加插件，除非你想改变在父代中定义的设置，否则不需要进行配置。

## Starters

Starters是一套方便的依赖描述符，可以包含在应用程序中。 您可以获得所需的所有Spring及相关技术的一站式服务，无需搜索示例代码，也不需要粘贴大量依赖描述符。 例如，如果你想开始使用Spring和JPA来访问数据库，只需在你的项目中加入spring-boot-starter-data-jpa依赖项，你就可以开始了。

初学者包含很多依赖项，您需要快速启动并运行一个项目，并使用一组支持的传递依赖项。

|  |
| --- |
| **What’s in a name**  所有官方首发者都遵循类似的命名模式; spring-boot-starter- \*，其中\*是特定类型的应用程序。 这种命名结构旨在帮助您找到启动器。 许多IDE中的Maven集成允许您按名称搜索依赖项。 例如，安装适当的Eclipse或STS插件后，只需在POM编辑器中点击ctrl-space并键入“spring-boot-starter”即可获得完整列表。  正如创建自己的启动器([Creating your own starter](#_bookmark292) )部分中所解释的，第三方启动器不应该以spring-boot为开始，因为它是为官方Spring Boot工件保留的。 acme的第三方初学者通常被命名为acme-spring-boot-starter。 |

下面的应用程序启动器是由Spring Boot提供的

org.springframework.boot group:

*Table 13.1. Spring Boot application starters*

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Pom** |
| spring-boot-starter | 核心启动器，包括自动启动器， | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter/pom.xml) |
|  | 配置支持，日志记录 |  |
|  | 和YAML |  |
| spring-boot-starter- activemq | 使用Apache ActiveMQ启动JMS消息传递 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-activemq/pom.xml) |
| spring-boot-starter- amqp | 使用Spring AMQP和Rabbitmq | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-amqp/pom.xml) |
| spring-boot-starter-aop | 面向切面 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-aop/pom.xml) |
|  | 用Spring AOP编程 |  |
|  | 和AspectJ |  |
| spring-boot-starter- | JMS消息传递 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-artemis/pom.xml) |
| artemis | 使用Apache Artemis |  |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Pom** |
| spring-boot-starter- | Starter for using Spring Batch | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-batch/pom.xml) |
| batch |  |  |
| spring-boot-starter- | Starter for using Spring | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-cache/pom.xml) |
| cache | Framework’s caching support |  |
| spring-boot-starter- cloud-connectors | Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-cloud-connectors/pom.xml) |
| spring-boot-starter- data-cassandra | Starter for using Cassandra distributed database and Spring Data Cassandra | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-cassandra/pom.xml) |
| spring-boot-starter- data-couchbase | Starter for using Couchbase document-oriented database and Spring Data Couchbase | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-couchbase/pom.xml) |
| spring-boot-starter- data-elasticsearch | Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-elasticsearch/pom.xml) |
| spring-boot-starter- data-gemfire | Starter for using GemFire distributed data store and Spring Data GemFire | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-gemfire/pom.xml) |
| spring-boot-starter- | Starter for using Spring Data | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-jpa/pom.xml) |
| data-jpa | JPA with Hibernate |  |
| spring-boot-starter- | Starter for using Spring Data | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-ldap/pom.xml) |
| data-ldap | LDAP |  |
| spring-boot-starter- data-mongodb | Starter for using MongoDB document-oriented database and Spring Data MongoDB | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-mongodb/pom.xml) |
| spring-boot-starter- data-neo4j | Starter for using Neo4j graph database and Spring Data Neo4j | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-neo4j/pom.xml) |
| spring-boot-starter- data-redis | Starter for using Redis key- value data store with Spring Data Redis and the Jedis client | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-redis/pom.xml) |
| spring-boot-starter- data-rest | Starter for exposing Spring Data repositories over REST using Spring Data REST | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-rest/pom.xml) |
| spring-boot-starter- data-solr | Starter for using the Apache Solr search platform with Spring Data Solr | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-data-solr/pom.xml) |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Pom** |
| spring-boot-starter- freemarker | Starter for building MVC web applications using FreeMarker views | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-freemarker/pom.xml) |
| spring-boot-starter- groovy-templates | Starter for building MVC web applications using Groovy Templates views | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-groovy-templates/pom.xml) |
| spring-boot-starter- hateoas | Starter for building hypermedia- [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-hateoas/pom.xml) based RESTful web application  with Spring MVC and Spring HATEOAS | |
| spring-boot-starter- integration | Starter for using Spring Integration | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-integration/pom.xml) |
| spring-boot-starter- jdbc | Starter for using JDBC with the [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jdbc/pom.xml) Tomcat JDBC connection pool | |
| spring-boot-starter- | Starter for building RESTful | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jersey/pom.xml) |
| jersey | web applications using JAX-RS | |
|  | and Jersey. An alternative to | |
|  | [spring-boot-starter-web](#_bookmark47) | |
| spring-boot-starter- | Starter for using jOOQ to | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jooq/pom.xml) |
| jooq | access SQL databases. An | |
|  | alternative to [spring-boot-](#_bookmark45) | |
|  | [starter-data-jpa](#_bookmark45) or | |
|  | [spring-boot-starter-](#_bookmark46) | |
|  | [jdbc](#_bookmark46) | |
| spring-boot-starter- | Starter for JTA transactions | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jta-atomikos/pom.xml) |
| jta-atomikos | using Atomikos | |
| spring-boot-starter- | Starter for JTA transactions | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jta-bitronix/pom.xml) |
| jta-bitronix | using Bitronix | |
| spring-boot-starter- jta-narayana | Spring Boot Narayana JTA Starter | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jta-narayana/pom.xml) |
| spring-boot-starter- mail | Starter for using Java Mail and Spring Framework’s email sending support | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-mail/pom.xml) |
| spring-boot-starter- mobile | Starter for building web applications using Spring Mobile | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-mobile/pom.xml) |
| spring-boot-starter- mustache | Starter for building MVC web applications using Mustache views | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-mustache/pom.xml) |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Pom** |
| spring-boot-starter- | Starter for using Spring Security [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-security/pom.xml) | |
| security |  | |
| spring-boot-starter- | Starter for using Spring Social | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-social-facebook/pom.xml) |
| social-facebook | Facebook | |
| spring-boot-starter- | Stater for using Spring Social | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-social-linkedin/pom.xml) |
| social-linkedin | LinkedIn | |
| spring-boot-starter- | Starter for using Spring Social | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-social-twitter/pom.xml) |
| social-twitter | Twitter | |
| spring-boot-starter- | Starter for testing Spring Boot | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-test/pom.xml) |
| test | applications with libraries | |
|  | including JUnit, Hamcrest and | |
|  | Mockito | |
| spring-boot-starter- thymeleaf | Starter for building MVC web applications using Thymeleaf views | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-thymeleaf/pom.xml) |
| spring-boot-starter- validation | Starter for using Java Bean Validation with Hibernate Validator | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-validation/pom.xml) |
| spring-boot-starter-web | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-web/pom.xml) |
| spring-boot-starter- web-services | Starter for using Spring Web Services | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-web-services/pom.xml) |
| spring-boot-starter- | Starter for building WebSocket [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-websocket/pom.xml) | |
| websocket | applications using Spring | |
|  | Framework’s WebSocket | |
|  | support | |

In addition to the application starters, the following starters can be used to add [*production ready*](#_bookmark297) features:

*Table 13.2. Spring Boot production starters*

|  |  |
| --- | --- |
| **Name**  spring-boot-starter- actuator | **Description Pom**  Starter for using Spring Boot’s [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-actuator/pom.xml) Actuator which provides  production ready features to help you monitor and manage your application |
| spring-boot-starter- remote-shell | Starter for using the CRaSH [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-remote-shell/pom.xml) remote shell to monitor and |

|  |  |
| --- | --- |
| **Name** | **Description Pom** |
|  | manage your application over SSH. Deprecated since 1.5 |

Finally, Spring Boot also includes some starters that can be used if you want to exclude or swap specific technical facets:

*Table 13.3. Spring Boot technical starters*

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Pom** |
| spring-boot-starter- jetty | Starter for using Jetty as the embedded servlet container. An alternative to [spring-boot-](#_bookmark49) [starter-tomcat](#_bookmark49) | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-jetty/pom.xml) |
| spring-boot-starter- log4j2 | Starter for using Log4j2 for logging. An alternative to | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-log4j2/pom.xml) |
|  | [spring-boot-starter-](#_bookmark48) [logging](#_bookmark48) | |
| spring-boot-starter- logging | Starter for logging using Logback. Default logging starter | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-logging/pom.xml) |
| spring-boot-starter- | Starter for using Tomcat as the | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-tomcat/pom.xml) |
| tomcat | embedded servlet container. | |
|  | Default servlet container starter | |
|  | used by [spring-boot-](#_bookmark47) | |
|  | [starter-web](#_bookmark47) | |
| spring-boot-starter- undertow | Starter for using Undertow as the embedded servlet container. An alternative to | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-starters/spring-boot-starter-undertow/pom.xml) |
|  | [spring-boot-starter-](#_bookmark49) [tomcat](#_bookmark49) | |

**Tip**

有关其他社区贡献者的列表，请参阅GitHub上的spring-boot-starters模块中的[README file](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/README.adoc) 。

# 构建你的 code

Spring Boot不需要任何特定的代码布局，但是，有一些最佳实践可以提供帮助。

## 使用 默认的 package

当一个类不包含包声明时，它被认为是在“默认包”中。 通常不鼓励使用“默认软件包”，应该避免使用“默认软件包”。 对于使用@ComponentScan，@EntityScan或@SpringBootApplication注释的Spring Boot应用程序来说，这可能会导致特定的问题，因为每个jar的每个类都将被读取。

**Tip**

我们建议您遵循Java推荐的软件包命名约定，并使用反向域名（例如com.example.project）。

## 加载 main application class

我们通常建议您将主应用程序类放在其他类的根包中。 @EnableAutoConfiguration注释通常放在主类上，它隐式地为某些项目定义了一个基本的“搜索包”。 例如，如果您正在编写JPA应用程序，则@EnableAutoConfiguration注释类的包将用于搜索@Entity项目。

使用根包也允许使用@ComponentScan注释而不需要指定basePackage属性。 如果您的主类位于根包中，也可以使用@SpringBootApplication注释。

这是一个典型的布局：

com

+- example

+- myproject

+- Application.java

|

+- domain

| +- Customer.java

| +- CustomerRepository.java

|

+- service

| +- CustomerService.java

|

+- web

+- CustomerController.java

The Application.java file would declare the main method, along with the basic @Configuration.

**package** com.example.myproject;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

@Configuration @EnableAutoConfiguration @ComponentScan

**public class** Application {

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

}

}

# 配置 classes

Spring Boot支持基于Java的配置。 虽然可以使用XML源调用SpringApplication.run（），但我们通常建议您的主要来源是@Configuration类。 通常，定义主要方法的类也是主要@Configuration的一个好选择。

**Tip**

在互联网上已经发布了许多使用XML配置的Spring配置示例。 如果可能，请始终尝试使用基于Java的等效配置。 搜索Enable\*注释可以是一个很好的起点。

## 导入其他配置类

你不需要把所有的@Configuration放到一个类中。 @Import注释可用于导入其他配置类。 或者，您可以使用@ComponentScan自动获取所有Spring组件，包括@Configuration类。.

## 导入 XML 配置

如果您绝对必须使用基于XML的配置，我们建议您仍以@Configuration类开头。 然后您可以使用额外的@ImportResource注解来加载XML配置文件。

# Auto-configuration

Spring Boot自动配置会尝试根据您添加的jar依赖项自动配置您的Spring应用程序。 例如，如果HSQLDB在您的类路径中，并且您没有手动配置任何数据库连接Bean，那么我们将自动配置一个内存数据库。

您需要通过添加@EnableAutoConfiguration或者选择加入自动配置

@SpringBootApplication 或 @Configuration 。.

**Tip**

您应该只添加一个@EnableAutoConfiguration注释。 我们通常建议您将其添加到您的主要@Configuration类。

## 主键替代 auto-configuration

自动配置是非侵入性的，在任何时候您都可以开始定义自己的配置来替换自动配置的特定部分。 例如，如果添加自己的DataSource bean，则默认的嵌入式数据库支持将退出。

如果您需要了解当前正在应用的自动配置，以及为什么使用--debug开关启动您的应用程序。 这将启用选择核心记录器的调试日志，并将自动配置报告记录到控制台。

## 禁用特定的 auto-configuration

如果您发现正在应用您不需要的特定自动配置类，则可以使用@EnableAutoConfiguration的exclude属性来禁用它们。

**import** org.springframework.boot.autoconfigure.\*; **import** org.springframework.boot.autoconfigure.jdbc.\*; **import** org.springframework.context.annotation.\*;

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

}

如果类不在类路径中，则可以使用注释的excludeName属性，并指定完全限定的名称。 最后，您还可以通过spring.autoconfigure.exclude属性来控制自动配置类的列表。

**Tip**

您可以在注释级别和使用属性中定义排除。

# Spring Beans 和依赖注入

您可以自由使用任何标准的Spring框架技术来定义您的bean及其注入的依赖关系。 为了简单起见，我们经常发现使用@ComponentScan来查找bean，并结合使用@Autowired构造函数注入效果很好。

如果按照上面的建议构建代码（在根包中定位应用程序类），则可以添加@ComponentScan而不带任何参数。 所有的应用程序组件（@Component，@Service，@Repository，@Controller等）都将被自动注册为Spring Bean。

这里是一个示例@Service Bean，它使用构造函数注入来获取所需的RiskAssessor bean。

**package** com.example.service;

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

**import** org.springframework.stereotype.Service;

@Service

**public class** DatabaseAccountService **implements** AccountService {

**private final** RiskAssessor riskAssessor; @Autowired

**public** DatabaseAccountService(RiskAssessor riskAssessor) {

**this**.riskAssessor = riskAssessor;

}

*// ...*

}

而如果一个bean有一个构造函数，你可以省略@Autowired。

@Service

**public class** DatabaseAccountService **implements** AccountService {

**private final** RiskAssessor riskAssessor;

**public** DatabaseAccountService(RiskAssessor riskAssessor) {

**this**.riskAssessor = riskAssessor;

}

*// ...*

}

**Tip**

请注意，如何使用构造函数注入允许riskAssessor字段被标记为final，表明它不能被随后更改。

# 使用 @SpringBootApplication 注解

许多Spring Boot开发人员总是使用@Configuration，@EnableAutoConfiguration和@ComponentScan注解其主类。 由于这些注释经常一起使用（特别是如果您遵循以上最佳实践[best practices](#_bookmark50) ），Spring Boot提供了一种方便的@SpringBootApplication替代方法。

@SpringBootApplication注释等价于使用@Configuration，@EnableAutoConfiguration和@ComponentScan及其默认属性：

**package** com.example.myproject;

**import** org.springframework.boot.SpringApplication;

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

@SpringBootApplication *// same as @Configuration @EnableAutoConfiguration @ComponentScan*

**public class** Application {

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

}

}

**Note**

@SpringBootApplication还提供为@EnableAutoConfiguration 和 @ComponentScan注解

别名定制的属性

# 运行你的应用程序

将应用程序打包为jar并使用嵌入式HTTP服务器的最大优点之一就是您可以像运行其他应用程序一样运行应用程序。 调试Spring Boot应用程序也很容易; 你不需要任何特殊的IDE插件或扩展。

**Note**

本节仅介绍基于jar的打包。如果您选择将应用程序打包为war文件，则应参考您的服务器和IDE文档。

## 从一个 IDE 运行

您可以从IDE运行Spring Boot应用程序作为简单的Java应用程序，但是，首先您需要导入项目。 导入步骤取决于您的IDE和构建系统。 大多数IDE可以直接导入Maven项目，例如Eclipse用户可以从File菜单中选择Import ...Existing Maven Projects。

如果不能直接将项目导入到IDE中，则可以使用构建插件生成IDE元数据。 Maven包含[Eclipse](http://maven.apache.org/plugins/maven-eclipse-plugin/) 和 [IDEA](http://maven.apache.org/plugins/maven-idea-plugin/)的插件; Gradle为各种IDE提供插件[various IDEs](https://docs.gradle.org/2.14.1/userguide/userguide.html).。

**Tip**

如果您不小心运行了两次Web应用程序，则会看到“端口已被使用”错误。 STS用户可以使用“重新启动”按钮而不是“运行”来确保关闭任何现有的实例。

## 打包运行应用程序

如果您使用Spring Boot Maven或Gradle插件创建可执行jar，则可以使用java -jar运行应用程序。 例如：

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

也可以运行打包的应用程序并启用远程调试支持。 这使您可以将调试器附加到打包的应用程序：

$ java -Xdebug -Xrunjdwp:server=y,transport=dt\_socket,address=8000,suspend=n \

-jar target/myproject-0.0.1-SNAPSHOT.jar

## 使用 Maven 插件

Spring Boot Maven插件包含一个可用于快速编译和运行应用程序的运行目标。 应用程序以分解形式运行，就像在IDE中一样。

$ mvn spring-boot:run

您可能还想使用有用的操作系统环境变量：

$ export MAVEN\_OPTS=-Xmx1024m -XX:MaxPermSize=128M

## 使用 Gradle 插件

Spring Boot Gradle插件还包含一个bootRun任务，可用于以分解形式运行您的应用程序。 无论何时导入spring-boot-gradle-plugin，都会添加bootRun任务：

$ gradle bootRun

您可能也想使用这个有用的操作系统环境变量：

$ export JAVA\_OPTS=-Xmx1024m -XX:MaxPermSize=128M

## 热部署

由于Spring Boot应用程序只是普通的Java应用程序，所以JVM热插拔应该是开箱即用的。 JVM热交换在某种程度上受限于它可以替换的字节码，为了获得更完整的解决方案，可以使用[JRebel](http://zeroturnaround.com/software/jrebel/) 或[Spring Loaded](https://github.com/spring-projects/spring-loaded) 项目。 spring-boot-devtools模块还包括对快速应用程序重新启动的支持。

有关详细信息，请参阅下面的第20章开发人员工具([Chapter 20, *Developer tools*](#_bookmark67) )部分和热插拔“操作方法”

([Hot swapping “How-to”](#_bookmark537) )。

# 开发者工具

Spring Boot包含一组额外的工具，可以使应用程序开发体验更愉快。 spring-boot-devtools模块可以包含在任何项目中以提供额外的开发时间功能。 要包含devtools支持，只需将模块依赖关系添加到您的构建：

**Maven.**

**<dependencies>**

**<dependency>**

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

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

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

**</dependency>**

**</dependencies>**

**Gradle.**

dependencies {

compile(***"org.springframework.boot:spring-boot-devtools"***)

}

**Note**

运行完整打包的应用程序时，开发人员工具会自动禁用 如果您的应用程序是使用java -jar启动的，或者如果它是使用特殊的类加载器启动的，那么它就被认为是“生产应用程序”。 将依赖关系标记为可选项是一种最佳实践，可以防止devtools通过项目传递到其他模块。 Gradle不支持可选的依赖关系，因此您可能希望在此期间看看[propdeps-plugin](https://github.com/spring-projects/gradle-plugins/tree/master/propdeps-plugin) 插件。

**Tip**

重新打包的档案在默认情况下不包含devtools。 如果您想使用某些远程devtools功能([certain remote devtools](#_bookmark79) [feature](#_bookmark79))，则需要禁用excludeDevtools构建属性以包含它。 该属性支持Maven和Gradle插件。

## 默认属性值

Spring Boot支持的一些库使用缓存来提高性能。 例如，模板引擎([template engines](#_bookmark138) )将缓存已编译的模板，以避免重复解析模板文件。 此外，Spring MVC可以在服务静态资源时将HTTP缓存头添加到响应中。

虽然缓存在生产中非常有益，但在开发过程中可能会产生反作用，使您无法看到应用程序中刚才所做的更改。 由于这个原因，spring-boot-devtools默认会禁用这些缓存选项。

缓存选项通常由您的application.properties文件中的设置进行配置。 例如，Thymeleaf提供了spring.thymeleaf.cache属性。 而不需要手动设置这些属性，spring-boot-devtools模块将自动应用合理的开发时间配置。

**Tip**

有关应用的属性的完整列表，请参阅[DevToolsPropertyDefaultsPostProcessor](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-devtools/src/main/java/org/springframework/boot/devtools/env/DevToolsPropertyDefaultsPostProcessor.java)。

## 自动重启

使用spring-boot-devtools的应用程序将在类路径上的文件发生更改时自动重启。 在IDE中工作时，这是一个非常有用的功能，因为它为代码更改提供了一个非常快速的反馈循环。 默认情况下，将监视指向文件夹的类路径中的任何条目以进行更改。 请注意，某些资源（如静态资产和视图模板）不需要重新启动应用程序(s [do not need to restart](#_bookmark71) [the application](#_bookmark71))。

|  |
| --- |
| **触发重启**  **由于DevTools监视类路径资源，触发重启的唯一方法是更新类路径。 导致类路径更新的方式取决于您使用的IDE。 在Eclipse中，保存修改后的文件将导致类路径更新并触发重启。 在IntelliJ IDEA中，构建项目（Build - > Make Project）将具有相同的效果。** |

**Note**

您也可以通过受支持的构建插件（即Maven和Gradle）启动您的应用程序，只要启用了分叉功能，因为DevTools需要隔离的应用程序类加载器才能正常运行。 当Gradle和Maven在类路径中检测到DevTools时，默认会这样做。

**Tip**

与LiveReload一起使用时，自动重新启动的效果非常好。 详情请参阅下文([See below](#_bookmark77) )。 如果使用JRebel，自动重新启动将被禁用，以支持动态类重新加载。 其他devtools功能（如LiveReload和属性覆盖）仍然可以使用。

**Note**

DevTools依靠应用程序上下文的关闭挂起在重新启动期间关闭它。 如果您禁用了关闭挂起（SpringApplication.setRegisterShutdownHook（false）），它将无法正常工作。

**Note**

当确定类路径上的条目在更改时会触发重新启动时，DevTools会自动忽略名为spring-boot，spring-boot-devtools，spring-boot-autoconfigure，spring-boot-actuator和spring-boot-starter的项目。

**Note**

DevTools需要自定义ApplicationContext使用的ResourceLoader：如果你的应用程序已经提供了一个，它将被打包。 不支持直接覆盖ApplicationContext上的getResource方法。

|  |
| --- |
| 重新启动vs重新加载  Spring Boot提供的重启技术通过使用两个类加载器来工作。 不改变的类（例如来自第三方jar的类）被加载到基类加载器中。 您正在开发的类将加载到重启类加载器中。 当应用程序重新启动时，重启classloader被丢弃，并创建一个新的。 这种方法意味着应用程序重新启动通常比“冷启动”快得多，因为基类加载器已经可用并且已经被填充了。  如果您发现重新启动对于您的应用程序来说不够快，或者遇到类加载问题，则可以考虑从ZeroTurnaround中重新加载技术，例如[JRebel](http://zeroturnaround.com/software/jrebel/) 。 这些工作通过重写类，因为他们被加载，使他们更容易重新加载。 [Spring](https://github.com/spring-projects/spring-loaded) [Loaded](https://github.com/spring-projects/spring-loaded)提供了另一种选择，但是它不支持许多框架，并且没有商业支持。 |

### 排除资源

某些资源不一定需要在更改时触发重新启动。 例如，Thymeleaf模板可以就地编辑。 默认情况下，更改/ META-INF / maven，/ META-INF / resources，/ resources，/ static，/ public或/ templates中的资源不会触发重新启动，但会触发实时重新加载([live reload](#_bookmark77))。 如果你想自定义这些排除，你可以使用spring.devtools.restart.exclude属性。 例如，要仅排除/ static和/ public，您可以设置以下内容：

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

**Tip**

如果你想保持这些默认值，并添加额外的排除，使用

spring.devtools.restart.additional-exclude属性。

### 监听额外的路径

您可能希望在更改不在类路径中的文件时重新启动或重新加载应用程序。 为此，请使用spring.devtools.restart.additional- paths属性来配置其他路径以监视更改。 您可以使用上述的spring.devtools.restart.exclude属性来控制额外路径[described above](#_bookmark71)下的更改是否会触发完全重新启动或仅实时重新加载[live reload](#_bookmark77).。

### 禁用重启

如果您不想使用重新启动功能，则可以使用spring.devtools.restart.enabled属性将其禁用。 在大多数情况下，你可以在你的application.properties中设置它（这将仍然初始化重启类加载器，但它不会监视文件的变化）。

如果您需要完全禁用重新启动支持，例如，因为它不适用于特定的库，则需要在调用SpringApplication.run（...）之前设置System属性。 例如

**public static void** main(String[] args) { System.setProperty(***"spring.devtools.restart.enabled"***, ***"false"***); SpringApplication.run(MyApp.**class**, args);

}

### 使用一个触发文件

如果您使用连续编译已更改文件的IDE，则可能只希望在特定时间触发重新启动。 要做到这一点，你可以使用“触发文件”，这是一个特殊的文件，当你想要实际触发重新启动检查时必须修改。 只更改文件会触发检查，只有Devtools检测到必须执行某些操作时才会重新启动。 触发文件可以手动更新，也可以通过IDE插件进行更新。

要使用触发器文件，请使用spring.devtools.restart.trigger-file属性。

**Tip**

您可能需要将spring.devtools.restart.trigger-file设置为全局设置([global setting](#_bookmark78) )，以便所有项目的行为方式相同。

### 自定义重启类 classloader

如上面的[Restart vs Reload](#_bookmark70)部分所述，重新启动功能是通过使用两个类加载器来实现的。 对于大多数应用程序来说，这种方法运行良好，但有时会导致类加载问题。

默认情况下，IDE中的任何打开的项目都将使用“重新启动”类加载器以及任何常规的加载器加载

.jar文件将使用“base”类加载器加载。 如果您使用多模块项目，而不是将每个模块导入到IDE中，

则可能需要自定义项目。 要做到这一点，你可以创建一个META-INF / spring-devtools.properties文件。

spring-devtools.properties文件可以包含restart.exclude。 和restart.include。 前缀属性。 include元素是应该被拉入到“重启”类加载器中的项目，排除元素是应该被下推到“基本”类加载器中的项目。 该属性的值是一个将应用于类路径的正则表达式模式。

例如:

**restart.exclude.companycommonlibs**=/mycorp-common-[\\w-]+\.jar **restart.include.projectcommon**=/mycorp-myproj-[\\w-]+\.jar

**Note**

所有的配置钥匙必须是唯一的。 只要属性以restart.include开头。 要么

restart.exclude。 这将被考虑。

**Tip**

所有类路径中的META-INF / spring-devtools.properties都将被加载。 您可以将文件打包到项目中，也可以打包到项目使用的库中。

### 已知限制

对于使用标准ObjectInputStream进行反序列化的对象，重新启动功能无法正常工作。 如果您需要反序列化数据，则可能需要将Spring的ConfigurableObjectInputStream与Thread.currentThread（）。getContextClassLoader（）一起使用。

不幸的是，有些第三方库反序列化，而不考虑上下文类加载器。 如果您发现这样的问题，您需要向原作者请求修复。

## 实时重新加载

spring-boot-devtools模块包含一个嵌入式LiveReload服务器，当资源发生变化时，可用于触发浏览器刷新。 LiveReload浏览器扩展可从[livereload.com](http://livereload.com/extensions/)的Chrome，Firefox和Safari免费获得。

如果您不想在应用程序运行时启动LiveReload服务器，则可以设置

spring.devtools.livereload.enabled属性为false。

**Note**

一次只能运行一个LiveReload服务器。 在开始您的应用程序之前，请确保没有其他LiveReload服务器正在运行。 如果您从IDE启动多个应用程序，则只有第一个应用程序支持LiveReload。

## 全局设置

您可以通过将一个名为.spring-boot- devtools.properties的文件添加到您的$ HOME文件夹来配置全局devtools设置（请注意文件名以“.”开头）。 添加到此文件的任何属性都将应用于使用devtools的计算机上的所有Spring Boot应用程序。 例如，要将重新启动配置为始终使用触发器文件([trigger file](#_bookmark74))，可以添加以下内容：

**~/.spring-boot-devtools.properties.**

**spring.devtools.reload.trigger-file**=.reloadtrigger

## 远程应用程序

Spring Boot开发人员工具不仅限于本地开发。 您还可以在远程运行应用程序时使用多个功能。 远程支持是可选的，为了启用它，您需要确保devtools包含在重新打包的归档中：

**<build>**

**<plugins>**

**<plugin>**

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

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

**<configuration>**

**<excludeDevtools>**false**</excludeDevtools>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

然后你需要设置一个spring.devtools.remote.secret属性，例如

**spring.devtools.remote.secret**=mysecret

**Warning**

在远程应用程序上启用spring-boot-devtools存在安全风险。 您不应该在生产部署上启用支持。

远程devtools支持分两部分提供; 有一个接受连接的服务器端点以及您在IDE中运行的客户端应用程序。 当设置了spring.devtools.remote.secret属性时，服务器组件会自动启用。 客户端组件必须手动启动。

### 运行远程客户端应用程序

远程客户端应用程序旨在从您的IDE中运行。 您需要使用与您要连接的远程项目相同的类路径来运行org.springframework.boot.devtools.RemoteSpringApplication。 传递给应用程序的非选项参数应该是您要连接到的远程URL。

例如，如果您使用的是Eclipse或STS，并且您已经将一个名为my-app的项目部署到Cloud Foundry，则可以执行以下操作：

* 从“运行”菜单中选择“运行配置...”。
* 创建一个新的Java应用程序“启动配置”。
* 浏览我的应用程序项目。
* 使用org.springframework.boot.devtools.RemoteSpringApplication作为主类。
* 将https://myapp.cfapps.io添加到程序参数（或任何远程URL）。

正在运行的远程客户端将如下所示：

Started RemoteSpringApplication in 0.74 seconds (JVM running for 1.105)

:

main] o.s.b.devtools.RemoteSpringApplication

:

main] o.s.b.d.a.OptionalLiveReloadServer

2015-06-10 18:25:07.074 INFO 14938 --- [

LiveReload server is running on port 35729 2015-06-10 18:25:07.130 INFO 14938 --- [

Starting RemoteSpringApplication on pwmbp with PID 14938 (/Users/pwebb/projects/spring-boot/code/ spring-boot-devtools/target/classes started by pwebb in /Users/pwebb/projects/spring-boot/code/spring- boot-samples/spring-boot-sample-devtools)

2015-06-10 18:25:06.671 INFO 14938 --- [ main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a17b7b6: startup date [Wed Jun 10 18:25:06 PDT 2015]; root of context hierarchy

2015-06-10 18:25:07.043 WARN 14938 --- [ main] o.s.b.d.r.c.RemoteClientConfiguration : The

connection to http://localhost:8080 is insecure. You should use a URL starting with 'https://'.

main] o.s.b.devtools.RemoteSpringApplication :

2015-06-10 18:25:06.632 INFO 14938 --- [

' | | . |\_| |\_|\_| |\_\ , | |\_|\_\ |\_|\_|\_\ /\ \ |/ / / /

=========|\_|==============| /===================================/\_/\_/\_/

:: Spring Boot Remote :: 1.5.8.RELEASE

/ -\_) ' \/ \_ \ \_/ -\_) ) ) ) )

| \_ \ \_ | |\_ \ \ \ \

\\/ )| |\_)| | | | | || (\_| []::::::[]

\_ \_

\ \ \ \

\_

/\\ / '\_ \_ \_(\_)\_ \_

( ( )\ | '\_ | '\_| | '\_ \/ \_` |

\_

.

**Note**

由于远程客户端使用与实际应用程序相同的类路径，因此可以直接读取应用程序属性。 这是如何读取spring.devtools.remote.secret属性并将其传递给服务器进行身份验证。

**Tip**

总是建议使用https：//作为连接协议，以便流量被加密，密码不能被拦截。

**Tip**

如果您需要使用代理来访问远程应用程序，请配置spring.devtools.remote.proxy.host和spring.devtools.remote.proxy.port属性。

### 远程更新

远程客户端将以与本地重启相同的方式监视您的应用程序类路径的更改。 任何更新的资源将被推送到远程应用程序，并（如果需要）触发重新启动。 如果您正在迭代使用您本地没有的云服务的功能，这可能会非常有帮助。 通常远程更新和重新启动比完整的重建和部署周期快得多。

**Note**

只有远程客户端正在运行时才监视文件。 如果在启动远程客户端之前更改文件，则不会将其推送到远程服务器。

### 远程调试隧道

Java远程调试在诊断远程应用程序的问题时非常有用。 不幸的是，当您的应用程序部署在数据中心之外时，并不总是可以启用远程调试。 如果您使用基于容器的技术（例如Docker），则远程调试也可能会非常棘手。

为了帮助解决这些限制，devtools支持通过HTTP隧道传输远程调试流量。 远程客户端在端口8000上提供本地服务器，您可以将其附加到远程调试器。 建立连接后，调试流量将通过HTTP发送到远程应用程序。 如果你想使用不同的端口，你可以使用spring.devtools.remote.debug.local-port属性。

您需要确保您的远程应用程序在启用远程调试的情况下启动。 这通常可以通过配置JAVA\_OPTS来实现。 例如，使用Cloud Foundry，您可以将以下内容添加到您的manifest.yml中：

*---*

**env**:

**JAVA\_OPTS**: ***"-Xdebug -Xrunjdwp:server=y,transport=dt\_socket,suspend=n"***

**Tip**

请注意，您不需要将address = NNNN选项传递给-Xrunjdwp。 如果省略，Java将简单地选取一个随机空闲端口。

**Note**

通过Internet调试远程服务可能会很慢，您可能需要增加IDE中的超时。 例如，在Eclipse中，您可以选择Java从首选项调试...并将调试器超时（ms）更改为更合适的值（在大多数情况下，60000可以正常工作）。

**Warning**

在IntelliJ IDEA中使用远程调试通道时，必须将所有断点配置为挂起线程而不是VM。 默认情况下，IntelliJ IDEA中的断点会挂起整个虚拟机，而不是挂起命中断点的线程。 这具有暂停管理远程调试通道的线程的不良副作用，导致您的调试会话冻结。 在IntelliJ IDEA中使用远程调试通道时，应将所有断点配置为挂起线程而不是VM。 请参阅[IDEA-165769](https://youtrack.jetbrains.com/issue/IDEA-165769) 了解更多详情。

# 打包生产环境中的应用程序

可执行的jar可用于生产部署。 由于它们是独立的，因此它们也非常适合基于云的部署。

对于额外的“生产就绪”功能，如健康，审计和度量REST或JMX端点; 考虑加入spring-boot-actuator执行器。 有关详细信息，请参见[*Part V, “Spring Boot Actuator: Production-*](#_bookmark297)[*ready features”*](#_bookmark297) for details。

# 接下来读什么

您现在应该对如何使用Spring Boot以及您应该遵循的一些最佳实践有很好的理解。 您现在可以继续深入了解特定的[*Spring Boot features*](#_bookmark85) ，或者可以跳过并阅读Spring Boot的“[production ready](#_bookmark297)”部分。

**Part IV. Spring Boot 特性**

本节将深入探讨Spring Boot的细节。 在这里，您可以了解要使用和定制的关键功能。 如果还没有，可能需要阅读[*Part II, “Getting started”*](#_bookmark8) *和* [*Part III, “Using Spring Boot”*](#_bookmark34) ，以便您了解基础知识。

# SpringApplication

SpringApplication类提供了一个方便的方法来引导将从main（）方法启动的Spring应用程序。 在很多情况下，你可以委托给静态的SpringApplication.run方法：

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

}

当您的应用程序启动时，您应该看到类似于以下内容：

Started SampleApplication in 2.992 seconds (JVM running for 3.658)

:

main] o.s.b.s.app.SampleApplication

main] .t.TomcatEmbeddedServletContainerFactory :

2014-03-04 13:09:54.912 INFO 41370 --- [

Server initialized with port: 8080

2014-03-04 13:09:56.501 INFO 41370 --- [

Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/myapp.jar started by pwebb) 2013-07-31 00:08:16.166 INFO 56603 --- [ main] ationConfigEmbeddedWebApplicationContext :

Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@6e5a8246:

startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy

:

main] o.s.b.s.app.SampleApplication

/\\ / '\_ \_ \_(\_)\_ \_ \ \ \ \ ( ( )\ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ )| |\_)| | | | | || (\_| | ) ) ) ) ' | | . |\_| |\_|\_| |\_\ , | / / / /

=========|\_|==============| /=/\_/\_/\_/

:: Spring Boot :: v1.5.8.RELEASE

2013-07-31 00:08:16.117 INFO 56603 --- [

\_ \_

\_

.

默认情况下会显示INFO日志消息，包括一些相关的启动详细信息，例如启动应用程序的用户。

## 启动失败

如果您的应用程序无法启动，已注册的FailureAnalyzers将有机会提供专门的错误消息和具体操作来解决问题。 例如，如果您在端口8080上启动Web应用程序，并且该端口已被使用，则应该看到类似于以下内容的内容：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* APPLICATION FAILED TO START

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Description:

Embedded servlet container failed to start. Port 8080 was already in use. Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.

**Note**

Spring Boot提供了大量的FailureAnalyzer实现，你可以很容易地添加你自己的([add your own](#_bookmark440))。

如果没有故障分析仪能够处理异常，您仍然可以显示完整的自动配置报告，以更好地了解出了什么问题。 所以你需要启用调试财产或启用调试日志记录

org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitialize

例如，如果使用java -jar运行应用程序，则可以按如下所示启用调试属性：

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

## 自定义 the Banner

启动时打印的Banner可以通过将banner.txt文件添加到类路径中或通过将banner.location设置为该文件的位置来更改。 如果文件有一个不寻常的编码，你可以设置banner.charset（默认是UTF-8）。 除了文本文件之外，还可以将banner.gif，banner.jpg或banner.png图像文件添加到您的类路径中，或设置banner.image.location属性。 图像将被转换成ASCII艺术表现形式并打印在任何文字横幅上方。

在您的banner.txt文件中，您可以使用以下任何占位符：

*Table 23.1. Banner variables*

|  |  |
| --- | --- |
| **Variable** | **Description** |
| ${application.version} | The version number of your application as declared in MANIFEST.MF. For example Implementation-Version: 1.0 is printed as 1.0. |
| ${application.formatted-version} | The version number of your application as declared in MANIFEST.MF formatted for display (surrounded with brackets and prefixed with v). For example (v1.0). |
| ${spring-boot.version}  ${spring-boot.formatted-version} | The Spring Boot version that you are using. For example 1.5.8.RELEASE.  The Spring Boot version that you are using formatted for display (surrounded with brackets and prefixed with v). For example (v1.5.8.RELEASE). |
| ${Ansi.NAME} (or ${AnsiColor.NAME},  ${AnsiBackground.NAME},  ${AnsiStyle.NAME}) | Where NAME is the name of an ANSI escape code. See [AnsiPropertySource](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/ansi/AnsiPropertySource.java) for details. |
| ${application.title} | The title of your application as declared in MANIFEST.MF. For example  Implementation-Title: MyApp is printed  as MyApp. |

**Tip**

如果要以编程方式生成Banner，则可以使用SpringApplication.setBanner（...）方法。 使用org.springframework.boot.Banner接口并实现自己的printBanner（）方法。

您还可以使用spring.main.banner-mode属性来确定banner是否必须使用已配置的logger（log）或根本不打印（关闭）打印在System.out（控制台）上。

打印的banner将以springBootBanner名称注册为单例bean。

**Note**

YAML映射为false，因此如果要禁用应用程序中的banner，请确保添加引号。

**spring**:

**main**:

**banner-mode**: ***"off"***

## 自定义 SpringApplication

如果SpringApplication默认不符合您的喜好，您可以创建一个本地实例并对其进行自定义。 例如，要关闭横幅，你会写：

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

SpringApplication app = **new** SpringApplication(MySpringConfiguration.**class**); app.setBannerMode(Banner.Mode.OFF);

app.run(args);

}

**Note**

传递给SpringApplication的构造函数参数是spring bean的配置源。 在大多数情况下，这些将是对@Configuration类的引用，但也可能是对XML配置或应扫描的包的引用。

也可以使用application.properties文件来配置SpringApplication。 有关详细信息，请参阅第[*Chapter 24, Externalized Configuration*](#_bookmark97) 。

有关配置选项的完整列表，请参阅[SpringApplication Javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/SpringApplication.html)。

## Fluent builder API

如果您需要构建一个ApplicationContext层次结构（具有父/子关系的多个上下文），或者如果您只想使用“fluent”构建器API，则可以使用SpringApplicationBuilder。

SpringApplicationBuilder允许你链接多个方法调用，包括parent 和 child的方法，允许你创建一个层

次结构。

**new** SpringApplicationBuilder()

.sources(Parent.**class**)

.child(Application.**class**)

.bannerMode(Banner.Mode.OFF)

.run(args);

For example:

**Note**

创建ApplicationContext层次结构时有一些限制，例如 Web组件必须包含在child context，并且相同的环境将用于parent and child contexts。 有关完整的细节，请参阅[SpringApplicationBuilder Javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/builder/SpringApplicationBuilder.html) 。

## Application events and listeners

除了通常的Spring框架事件，比如[ContextRefreshedEvent](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/context/event/ContextRefreshedEvent.html)，SpringApplication发送一些额外的应用

程序事件。

**Note**

有些事件实际上是在创建ApplicationContext之前触发的，因此您不能将这些监听器注册为@Bean。 您可以通过SpringApplication.addListeners（...）或SpringApplicationBuilder.listeners（...）方法注册它们。

如果您希望自动注册这些监听器而不管创建应用程序的方式如何，您可以将META-INF / spring.factories文件添加到您的项目中，并使用org.springframework.context.ApplicationListener项引用您的监听器。

org.springframework.context.ApplicationListener=com.example.project.MyListener

应用程序事件按照以下顺序发送，就像您的应用程序运行一样：

1. ApplicationStartingEvent在运行开始时发送，但在除监听器和初始化程序的注册之外的任何处理之前发送。
2. 当在上下文中使用的环境是已知的，但在上下文创建之前，发送plicationEnvironmentPreparedEvent。
3. 一个ApplicationPreparedEvent只在刷新开始之前发送，但是在bean定义被载入之后。
4. 刷新后发送ApplicationReadyEvent，并处理任何相关的回调以指示应用程序已准备好为请求提供服务。
5. 如果启动时出现异常，则发送ApplicationFailedEvent。

**Tip**

您通常不需要使用应用程序事件，但可以方便地知道它们存在。 在内部，Spring Boot使用事件来处理各种任务。

## Web environment

SpringApplication将会尝试为你创建正确类型的ApplicationContext。 默认情况下，将使用AnnotationConfigApplicationContext或AnnotationConfigEmbeddedWebApplicationContext，具体取决于您是否正在开发Web应用程序。

用于确定“Web环境”的算法相当简单（基于少数类的存在）。 如果您需要覆盖默认值，则可以使用setWebEnvironment（boolean webEnvironment）。

也可以通过调用setApplicationContextClass（...）来完全控制ApplicationContext类型。

**Tip**

使用SpringApplication时，通常需要调用setWebEnvironment（false）在JUnit测试中。

## 访问应用程序参数

如果您需要访问传递给SpringApplication.run的应用程序参数（...）你可以注入一个org.springframework.boot.ApplicationArguments bean。 ApplicationArguments接口提供对原始String []参数以及解析的选项和非选项参数的访问：

**import** org.springframework.boot.\*

**import** org.springframework.beans.factory.annotation.\*

**import** org.springframework.stereotype.\*

@Component

**public class** MyBean {

@Autowired

**public** MyBean(ApplicationArguments args) { **boolean** debug = args.containsOption(***"debug"***); List<String> files = args.getNonOptionArgs();

*// if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]*

}

}

**Tip**

Spring Boot也将在Spring环境中注册一个CommandLinePropertySource。 这使您可以使用@Value注释来注入单个应用程序参数。

## 使用 ApplicationRunner 或 CommandLineRunner

如果你需要在SpringApplication启动后运行一些特定的代码，你可以实现ApplicationRunner或者CommandLineRunner接口。 两个接口都以相同的方式工作，并提供了一个将在SpringApplication.run（...）完成之前调用的单个运行方法。

CommandLineRunner接口作为一个简单的字符串数组提供对应用程序参数的访问，而ApplicationRunner使用上面讨论的ApplicationArguments接口。

**import** org.springframework.boot.\*

**import** org.springframework.stereotype.\*

@Component

**public class** MyBean **implements** CommandLineRunner {

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

*// Do something...*

}

}

还可以实现org.springframework.core.Ordered接口或使用org.springframework.core.annotation.Order注释，如果定义了多个CommandLineRunner或ApplicationRunner bean，则必须按特定顺序调用它们。

## 退出应用程序

每个SpringApplication都会向JVM注册一个shutdown hook，以确保ApplicationContext在退出时正常关闭。 所有标准的Spring生命周期回调（比如DisposableBean接口或者@PreDestroy注解）都可以使用。

另外，如果在调用SpringApplication.exit（）时想要返回特定的退出代码，那么，bean可以实现org.springframework.boot.ExitCodeGenerator接口来满足这个要求。 这个退出代码然后可以传递给System.exit（）以返回它作为状态代码。

@SpringBootApplication

**public class** ExitCodeApplication {

@Bean

**public** ExitCodeGenerator exitCodeGenerator() {

**return new** ExitCodeGenerator() { @Override

**public int** getExitCode() {

**return** 42;

}

};

}

**public static void** main(String[] args) { System.exit(SpringApplication

.exit(SpringApplication.run(ExitCodeApplication.**class**, args)));

}

}

另外，ExitCodeGenerator接口可能由异常实现。 遇到这样的异常时，Spring Boot将返回由实现的getExitCode（）方法提供的退出代码。

## Admin 功能

通过指定spring.application.admin.enabled属性，可以为应用程序启用与管理相关的功能。 这暴露了平台MBeanServer上的[SpringApplicationAdminMXBean](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/admin/SpringApplicationAdminMXBean.java) 。 您可以使用此功能远程管理您的Spring Boot应用程序。 这对于任何服务包装实现也是有用的。

**Tip**

如果您想知道应用程序在哪个HTTP端口上运行，请使用key获取该属性local.server.port。

**Note**

启用此功能时要小心，因为MBean公开了关闭应用程序的方法。

# 外部化配置

Spring Boot允许您将配置外部化，以便在不同的环境中使用相同的应用程序代码。 您可以使用属性文件，YAML文件，environment variables和command-line来外部化配置。 属性值可以使用@Value注解直接注入到bean中，可以通过Spring的Environment抽象来访问，也可以通过@ConfigurationProperties绑定到结构化对象([bound to](#_bookmark109) [structured objects](#_bookmark109) )。

Spring Boot使用了一个非常特殊的PropertySource命令，该命令旨在允许合理的重写值。 属性按以下顺序考虑：

1. 在您的主目录上开发[Devtools global settings properties](#_bookmark78)（当devtools处于活动状态时〜/ .spring-boot- devtools.properties）。
2. 测试中的[@TestPropertySource](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/test/context/TestPropertySource.html) 注释.
3. 测试时[@[SpringBootTest#properties](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/test/context/SpringBootTest.html)](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/test/context/TestPropertySource.html) 注释属性.
4. 命令行参数
5. 来自SPRING\_APPLICATION\_JSON的属性（嵌入在环境变量或系统属性中的内联JSON）
6. ServletConfig 初始化参数.
7. ServletContext 初始化参数.
8. 来自 java:comp/env JNDI 属性.
9. Java System properties (System.getProperties()).
10. OS 环境变量.
11. RandomValuePropertySource只具有随机的属性\*。
12. [Profile-specific application properties](#_bookmark101) outside of your packaged jar (application-

{profile}.properties and YAML variants)

1. [Profile-specific application properties](#_bookmark101) packaged inside your jar (application-

{profile}.properties and YAML variants)

1. Application properties outside of your packaged jar (application.properties and YAML variants).
2. Application properties packaged inside your jar (application.properties and YAML variants).
3. [@PropertySource](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotations on your @Configuration classes.

17.Default properties (specified using SpringApplication.setDefaultProperties).

为了提供一个具体的例子，假设你开发了一个使用name属性的@Component：

**import** org.springframework.stereotype.\*

**import** org.springframework.beans.factory.annotation.\*

@Component

**public class** MyBean {

@Value("${name}")

**private** String name;

*// ...*

}

在您的应用程序类路径中（例如，在您的jar中），您可以拥有一个application.properties，为名称提供一个合理的默认属性值。 在新环境中运行时，可以在jar外部提供一个application.properties来覆盖名称; 对于一次性测试，您可以使用特定的命令行开关（例如java -jar app.jar - name =“Spring”）启动。

**Tip**

可以使用环境变量在命令行上提供SPRING\_APPLICATION\_JSON属性。 例如在一个UN \* X shell中：

$ SPRING\_APPLICATION\_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar

在这个例子中，您将在Spring环境中以foo.bar = spam结尾。 您还可以在System变量中提供JSON作为spring.application.json：

$ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar

或命令行参数：

$ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'

或者作为JNDI变量java：comp / env / spring.application.json。

## 配置 random values

RandomValuePropertySource用于注入随机值（例如，注入秘密或测试用例）。 它可以产生integers，longs，uuids或strings，例如

**my.secret**=${random.value} **my.number**=${random.int} **my.bignumber**=${random.long} **my.uuid**=${random.uuid} **my.number.less.than.ten**=${random.int(10)} **my.number.in.range**=${random.int[1024,65536]}

random.int \*语法是OPEN value（，max）CLOSE其中OPEN，CLOSE是任何字符和值，max是整数。 如果提供最大值，则值是最小值，最大值是最大值（不包括）。

## 访问 command line properties

默认情况下，SpringApplication将转换任何命令行选项参数（以' -- '开头，例如 --server.port = 9000）添加到一个属性并将其添加到Spring环境。 如上所述，命令行属性总是优先于其他属性源。

如果您不希望将命令行属性添加到环境中，则可以使用禁用它们SpringApplication.setAddCommandLineProperties（false）。

## 应用程序 property files

SpringApplication将从以下位置的application.properties文件中加载属性，并将它们添加到Spring环境中：

1. 当前目录的A / config子目录.
2. 当前目录
3. 一个 classpath /config 包
4. classpath root

该列表按优先顺序排列（在列表中较高的位置定义的属性将覆盖在较低位置定义的属性）。

**Note**

您也可以使用[use YAML ('.yml') files](#_bookmark103) 文件替代“.properties”

如果您不喜欢application.properties作为配置文件名，则可以通过指定spring.config.name环境属性来切换到另一个。 您还可以使用spring.config.location环境属性（以逗号分隔的目录位置列表或文件路径）引用显式位置。

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

或

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

**Warning**

spring.config.name和spring.config.location很早就用来确定哪些文件必须被加载，因此必须将其定义为环境属性（通常是OS env，系统属性或命令行参数）。

如果spring.config.location包含目录（而不是文件），它们应该以/结尾(并且会在加载之前附加从spring.config.name生成的名称，包括配置文件特定的文件名。) 在spring.config.location中指定的文件按原样使用，不支持特定于配置文件的变体，并且将被任何特定于配置文件的特性覆盖。

配置位置按相反顺序搜索。 默认情况下，配置的位置是classpath：/，classpath：/ config /，

file：. / file：./ config /。 结果搜索顺序是：

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

当配置自定义配置位置时，除了默认位置之外，还会使用它们。 在默认位置之前搜索自定义位置。 例如，如果配置了自定义位置classpath：/ custom-config /，file：./ custom-config /，则搜索顺序变为：

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

此搜索顺序允许您在一个配置文件中指定默认值，然后在另一个配置文件中选择性地覆盖这些值。 您可以在默认位置之一的application.properties（或其他选择任何使用spring.config.name命名的）中为您的应用程序提供默认值。 这些默认值可以在运行时被置于其中一个自定义位置的不同文件覆盖。

**Note**

如果使用环境变量而非系统属性，则大多数操作系统不允许使用句点分隔的键名(period-separated key names)，但可以使用下划线（例如SPRING\_CONFIG\_NAME，而不是spring.config.name）。

**Note**

如果您正在容器中运行，那么可以使用JNDI属性（在java：comp / env中）或servlet上下文初始化参数，而不是使用环境变量或系统属性。

## Profile-specific properties

除了application.properties文件外，还可以使用命名约定application- {profile} .properties来定义特定于配置文件的属性。 如果没有设置活动配置文件（即，如果没有显式激活配置文件，则加载来自application-default.properties的属性），则环境具有一组默认配置文件（默认情况下为[缺省值]）。

Profile-specific properties 从标准application.properties的相同位置加载，配置文件特定的文件总是覆盖非特定的文件，而不管配置文件特定的文件是在打包的jar内部还是外部。

如果指定了多个配置文件，则应用最后一个赢取策略(last wins strategy applies)。 例如，由spring.profiles.active属性指定的配置文件会在通过SpringApplication API配置的配置文件之后添加，因此优先。

**Note**

如果您在spring.config.location中指定了任何文件，则不会考虑这些文件的特定于配置文件的变体。 如果您还想使用配置文件特定的属性，请使用spring.config.location中的目录。

## 属性中的占位符

application.properties中的值在使用时会通过现有的Environment进行过滤，因此您可以返回到以前定义的值（例如，从System properties）。

**app.name**=MyApp

**app.description**=${app.name} is a Spring Boot application

**Tip**

您也可以使用这种技术来创建现有Spring Boot属性的“short’”变体。 有关详细信息，请参见[*Section 72.4, “Use ‘short’ command line arguments”*](#_bookmark451) 。

## 使用 YAML 代替 Properties

[YAML](http://yaml.org/)是JSON的超集，因此是用于指定分层配置数据的非常方便的格式。 SpringApplication类将自动支持YAML作为属性的替代方法，只要在classpath中有[SnakeYAML](http://www.snakeyaml.org/) 库。

**Note**

如果您使用“Starters”，SnakeYAML将通过spring-boot-starter自动提供。

### 加载 YAML

Spring框架提供了两个方便的类，可以用来加载YAML文档。 YamlPropertiesFactoryBean将作为属性加载YAML，YamlMapFactoryBean将作为Map加载YAML。

例如，下面的YAML文件：

**environments**: **dev**:

**url**[: http://dev.bar.com](http://dev.bar.com/)

**name**: Developer Setup

**prod**:

**url**[: http://foo.bar.com](http://foo.bar.com/)

**name**: My Cool App

将被转化为这些属性：

**environments.dev.url**[=http://dev.bar.com](http://dev.bar.com/) **environments.dev.name**=Developer Setup **environments.prod.url**[=http://foo.bar.com](http://foo.bar.com/) **environments.prod.name**=My Cool App

YAML列表被表示为具有[index]解引用的属性键，例如这个YAML：

**my**:

**servers**:

* dev.bar.com
* foo.bar.com

将被转化为这些属性：

**my.servers[0]**=dev.bar.com **my.servers[1]**=foo.bar.com

要使用Spring DataBinder实用程序（这是@ConfigurationProperties所做的）绑定属性，需要在java.util.List（或Set）类型的目标Bean中拥有一个属性，并且您需要提供setter，或者 用一个可变的值初始化它，

例如 这将绑定到上面的属性

@ConfigurationProperties(prefix="my")

**public class** Config {

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

**public** List<String> getServers() {

**return this**.servers;

}

}

**Note**

当按照您所期望的方式重写将无法正常工作时，需要额外的注意。 在上面的例子中，当my.servers被重新定义在几个地方时，单个元素是被覆盖的目标，而不是列表。 为了确保具有更高优先级的PropertySource可以覆盖列表，您需要将其定义为单个属性：

**my**:

**servers**: dev.bar.com,foo.bar.com

### 在Spring环境中将YAML作为属性公开

YamlPropertySourceLoader类可以用来在Spring环境中将YAML作为一个PropertySource公开。 这使您可以使用熟悉的@Value注释和占位符语法来访问YAML属性。

### Multi-profile YAML 文件

您可以使用spring.profiles在单个文件中指定多个特定于配置文件的YAML文档键来指示文件何时适用。

例如：

**server**:

**address**: 192.168.1.100

*---*

**spring**:

**profiles**: development

**server**:

**address**: 127.0.0.1

*---*

**spring**:

**profiles**: production

**server**:

**address**: 192.168.1.120

在上面的示例中，如果开发配置文件处于活动状态，则server.address属性将为127.0.0.1。 如果开发和生产配置文件未启用，则属性的值将是192.168.1.100。

如果在应用程序上下文启动时没有显式激活，则默认配置文件被激活。 所以在这个YAML中，我们设置了security.user.password的一个值，它只在“default”配置文件中可用：

**server**: **port**: 8000

*---*

**spring**:

**profiles**: default

**security**: **user**:

**password**: weak

而在这个例子中，密码总是被设置，因为它没有附加到任何配置文件，并且必须根据需要在所有其他配置文件中明确地重置:

**server**: **port**: 8000

**security**: **user**:

**password**: weak

使用“spring.profiles”元素指定的Spring配置文件可以有选择地使用！ 字符。 如果为单个文档指定了否定配置文件和非non-negated配置文件，则至少有一个非non-negated配置文件必须匹配，且non-negated配置文件可能不匹配。

### YAML 缺点

YAML文件不能通过@PropertySource注解加载。 所以在需要以这种方式加载值的情况下，您需要使用一个属性文件。

### Merging YAML lists

正如我们上面看到的([we have seen above](#_bookmark104))，任何YAML内容最终都会转换为属性。 当通过配置文件覆盖“列表”属性时，该过程可能不直观。

例如，假设名称和描述属性默认为空的MyPojo对象。 让我们从FooProperties公开一个MyPojo列表：

@ConfigurationProperties("foo")

**public class** FooProperties {

**private final** List<MyPojo> list = **new** ArrayList<>();

**public** List<MyPojo> getList() {

**return this**.list;

}

}

考虑以下配置：

**foo**:

**list**:

**- name**: my name

**description**: my description

*---*

**spring**: **profiles**: dev

**foo**:

**list**:

**- name**: my another name

如果开发者配置文件不活动，FooProperties.list将包含一个如上定义的MyPojo条目。 如果启用配置文件启用，但列表仍将只包含一个条目（名称“我的另一个名称”和说明空）。 此配置不会将第二个MyPojo实例添加到列表中，并且不会合并这些项目。

当在多个配置文件中指定一个集合时，将使用具有最高优先级的集合（并且仅使用该集合）：

**foo**:

**list**:

* **name**: my name

**description**: my description

* **name**: another name

**description**: another description

*---*

**spring**: **profiles**: dev

**foo**:

**list**:

**- name**: my another name

在上面的例子中，考虑到dev配置文件是活动的，FooProperties.list将包含一个MyPojo条目

（名称为“我的另一个名称”，说明为空）。

## Type-safe 配置文件

使用@Value（“$ {property}”）注释来注入配置属性有时会非常麻烦，特别是如果您使用多个属性或者您的数据是分层的。 Spring Boot提供了另一种处理属性的方法，允许强类型的bean管理和验证应用程序的配置。

**package** com.example;

**import** java.net.InetAddress; **import** java.util.ArrayList; **import** java.util.Collections; **import** java.util.List;

**import** org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("foo")

**public class** FooProperties {

**private boolean** enabled;

**private** InetAddress remoteAddress;

**private final** Security security = **new** Security();

**public boolean** isEnabled() { ... }

**public void** setEnabled(**boolean** enabled) { ... }

**public** InetAddress getRemoteAddress() { ... }

**public void** setRemoteAddress(InetAddress remoteAddress) { ... }

**public** Security getSecurity() { ... }

**public static class** Security { **private** String username; **private** String password;

**private** List<String> roles = **new** ArrayList<>(Collections.singleton(***"USER"***));

**public** String getUsername() { ... }

**public void** setUsername(String username) { ... }

**public** String getPassword() { ... }

**public void** setPassword(String password) { ... }

**public** List<String> getRoles() { ... }

**public void** setRoles(List<String> roles) { ... }

}

}

上面的 POJO 定义了以下属性:

* foo.enabled, 默认为 false
* foo.remote-address, 与可以从String强制类型
* foo.security.username, 具有名称由属性名称确定的嵌套安全性。特别是返回类型没有在那里使用，可能是SecurityProperties
* foo.security.password
* foo.security.roles, 与String的集合

**Note**

getters和setter通常是强制的，因为绑定是通过标准的Java Beans属性描述符来完成的，就像在Spring MVC中一样。 有些情况下，可能会忽略setter:

* + Maps, 只要它们被初始化，就需要一个getter，但不一定是setter，因为它们可以被绑定器binder改变。
  + 集合和数组可以通过索引（通常使用YAML）或使用单个逗号分隔值（属性）来访问。 在后一种情况下，setter是强制性的。 我们建议始终为这种类型添加一个setter。 如果初始化一个集合，确保它不是不可变的（如上例）
  + 如果初始化嵌套的POJO属性（如上例中的Security域），则不需要setter。 如果您希望binder使用其默认构造函数即时创建实例，则需要一个setter。

有些人使用Project Lombok来自动添加getter和setter。 确保Lombok不会为这种类型生成任何特定的构造函数，因为它将被容器自动使用来实例化对象。

**Tip**

另外请参考 [differences between @Value and @ConfigurationProperties](#_bookmark114) 的区别.

您还需要列出要在@EnableConfigurationProperties中注册的属性类

注解：

@Configuration

@EnableConfigurationProperties(FooProperties.class)

**public class** MyConfiguration {

}

**Note**

当以这种方式注册@ConfigurationProperties bean时，该bean将具有常规名称：<prefix> - <fqn>，其中<prefix>是@ConfigurationProperties注释中指定的环境键前缀，<fqn>是 bean。 如果注释不提供任何前缀，则只使用bean的完全限定名称。

上例中的bean名称将是 foo-com.example.FooProperties.

即使上面的配置为FooProperties创建了一个常规的bean，我们也建议@ConfigurationProperties只处理环境，特别是不要从上下文中注入其他bean。 话虽如此，@EnableConfigurationProperties注释也会自动应用到您的项目中，以便任何现有的使用@ConfigurationProperties注释的bean都将从环境配置。 您可以通过确保FooProperties已经是一个bean来快速配置MyConfiguration：

@Component @ConfigurationProperties(prefix="foo") **public class** FooProperties {

*// ... see above*

}

SpringApplication外部YAML配置的这种配置特别适合：

*# application.yml*

**foo**:

**remote-address**: 192.168.1.1

**security**:

**username**: foo

**roles**:

* USER
* ADMIN

*# additional configuration as required*

要使用@ConfigurationProperties bean，您可以像其他任何bean一样注入它们。

@Service

**public class** MyService {

**private final** FooProperties properties; @Autowired

**public** MyService(FooProperties properties) {

**this**.properties = properties;

}

*//...*

@PostConstruct

**public void** openConnection() {

Server server = **new** Server(**this**.properties.getRemoteAddress());

*// ...*

}

}

**Tip**

使用@ConfigurationProperties还允许您生成元数据文件，IDE可以使用这些元数据文件为自己的密钥提供自动完成功能，详情请参阅附录B[Appendix B, *Configuration meta-*](#_bookmark571)[*data*](#_bookmark571) ，配置元数据附录。

### Third-party 配置

除了使用@ConfigurationProperties注解一个类，你也可以在public @Bean方法上使用它。 如果要将属性绑定到不受控制的第三方组件，这可能特别有用。

要从Environment属性配置一个bean，添加@ConfigurationProperties到它的bean注册：

@ConfigurationProperties(prefix = "bar") @Bean

**public** BarComponent barComponent() {

...

}

使用bar前缀定义的任何属性都将以类似于上面的FooProperties示例的方式映射到该BarComponent bean上。

### Relaxed binding

Spring Boot使用一些宽松的规则来将环境属性绑定到@ConfigurationProperties bean，所以不需要在Environment属性名称和bean属性名称之间完全匹配。 其中有用的常见示例包括虚线分隔（例如，上下文路径绑定到contextPath）和大写（例如PORT绑定到端口）环境属性。

例如，给定以下@ConfigurationProperties类：

@ConfigurationProperties(prefix="person")

**public class** OwnerProperties {

**private** String firstName;

**public** String getFirstName() {

**return this**.firstName;

}

**public void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

}

以下属性名称都可以使用：

*Table 24.1. relaxed binding*

|  |
| --- |
| **Property Note** |
| person.firstNameStandard camel case syntax. |

|  |
| --- |
| **Property Note** |
| person.first- Dashed notation, recommended for use in .properties and .yml files.  name  person.first\_namUenderscore notation, alternative format for use in .properties and .yml  files. |
| PERSON\_FIRST\_NAMUEpper case format. Recommended when using a system environment variables. |

### Properties 转换

当它绑定到@ConfigurationProperties bean时，Spring将试图强制外部应用程序属性为正确的类型。 如果您需要自定义类型转换，您可以提供一个ConversionService bean（带有bean id conversionService）或自定义属性编辑器（通过CustomEditorConfigurer bean）或自定义Converters（注释为@ConfigurationPropertiesBinding的bean定义）。

**Note**

由于此bean在应用程序生命周期中很早被请求，因此请确保限制ConversionService所使用的依赖项。 通常情况下，您需要的任何依赖项可能在创建时未完全初始化。 如果不需要配置密钥强制转换，并且只依赖使用@ConfigurationPropertiesBinding限定的自定义转换器，则可能需要重命名自定义的ConversionService。

### @ConfigurationProperties 验证

当Spring的@Validated批注注释时，Spring Boot将尝试验证@ConfigurationProperties类。 您可以直接在您的配置类上使用JSR-303的javax.validation约束条件注释。 只要确保符合JSR-303的实现在您的类路径上，然后将约束注释添加到您的字段中：

@ConfigurationProperties(prefix="foo") @Validated

**public class** FooProperties {

@NotNull

**private** InetAddress remoteAddress;

*// ... getters and setters*

}

为了验证嵌套属性的值，您必须将相关字段注释为@Valid来触发其验证。 例如，建立在上面的FooProperties示例上：

@ConfigurationProperties(prefix="connection") @Validated

**public class** FooProperties {

@NotNull

**private** InetAddress remoteAddress;

@Valid

**private final** Security security = **new** Security();

*// ... getters and setters*

**public static class** Security { @NotEmpty

**public** String username;

*// ... getters and setters*

}

}

你也可以通过创建一个名为configurationPropertiesValidator的bean定义来添加一个自定义的Spring Validator。 应该声明@Bean方法是静态的。 配置属性验证器在应用程序生命周期的早期就被创建，并将@Bean方法声明为静态，这样就可以在不需要实例化@Configuration类的情况下创建bean。 这避免了可能由于实例化提前而导致的任何问题。 有一个属性验证示例[property validation sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-property-validation) ，所以你可以看到如何设置的东西。

**Tip**

spring-boot-actuator模块包含一个暴露所有@ConfigurationProperties bean的端点。 只需将您的Web浏览器指向/ configprops或使用等效的JMX端点即可。 请参阅生产准备功能[*Production ready features*](#_bookmark299)。 部分的细节。

### @ConfigurationProperties vs. @Value

@Value是核心容器功能，它不提供与类型安全配置属性相同的功能。 下表总结了@ConfigurationProperties和@Value支持的功能：

|  |  |
| --- | --- |
| **Feature @ConfigurationProp** | **@Value** |
| [Relaxed binding](#_bookmark111) Yes  [Meta-data support](#_bookmark571) Yes | No  No |
| SpEL evaluation No | Yes |

如果您为自己的组件定义了一组配置关键字，我们建议您将它们分组到POJO注释的@ConfigurationProperties中。 另请注意，由于@Value不支持放宽绑定，因此如果您需要使用环境变量提供值，则不是一个好选择。

最后，虽然可以在@Value中编写SpEL表达式，但是这些表达式不会从应用程序属性文件中处理[Application property files](#_bookmark100)。

# Profiles

Spring Profiles提供了一种分离部分应用程序配置的方法，并使其仅在特定环境中可用。 任何@Component或@Configuration都可以使用@Profile标签来限制何时加载:

在普通的Spring方法中，可以使用spring.profiles.active环境属性来指定哪些配置文件处于活动状态。 你可以用任何常用的方式来指定属性，例如你可以在你的application.properties中包含它：

**spring.profiles.active**=dev,hsqldb

或使用 switch --spring.profiles.active = dev，hsqldb在命令行中指定。

## 添加激活 profiles

spring.profiles.active属性遵循与其他属性相同的排序规则，最高的PropertySource将获胜。 这意味着您可以在application.properties中指定活动配置文件，然后使用命令行开关替换它们。

有时，将特定于配置文件的属性添加到活动配置文件而不是替换它们会很有用。 spring.profiles.include属性可以用来无条件添加活动配置文件。 SpringApplication入口点还有一个用于设置其他配置文件的Java API（即，在由spring.profiles.active属性激活的那些配置文件之上）：请参阅setAdditionalProfiles（）方法。

例如，当具有以下属性的应用程序使用switch - spring.profiles.active = prod运行时，proddb和prodmq配置文件也将被激活：

*---*

**my.property**: fromyamlfile

*---*

**spring.profiles**: prod

**spring.profiles.include**:

* proddb
* prodmq

**Note**

请记住，可以在YAML文档中定义spring.profiles属性，以确定何时将此特定文档包含在配置中。 有关更多详细信息，请参见[Section 72.7,](#_bookmark454) [“Change configuration depending on the environment”](#_bookmark454) 。

## 变成方式配置 profiles

您可以通过在应用程序运行之前调用SpringApplication.setAdditionalProfiles（...）以编程方式设置活动配置文件。 使用Spring的ConfigurableEnvironment接口也可以激活配置文件。

## Profile-specific configuration files

将application.properties（或application.yml）和通过@ConfigurationProperties引用的文件的特定于配置文件的变体(Profile-specific variants)视为加载文件。 有关详细信息，请参见[*Section 24.4,*](#_bookmark101)[*“Profile-specific properties”*](#_bookmark101) 。

# Logging

Spring Boot使用[Commons Logging](http://commons.apache.org/logging) 进行所有内部日志记录，但是将底层日志实现保留为打开状态。 为[Java Util Logging](http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html), [Log4J2](http://logging.apache.org/log4j/2.x/) 和 [Logback](http://logback.qos.ch/)提供了默认配置。 在每种情况下，loggers都预先配置为使用控制台输出，可选的文件输出也可用。

默认情况下，如果使用“Starter”，Logback将用于日志记录。 还包括适当的Logback路由，以确保使用Java Util日志记录，Commons Logging，Log4J或SLF4J的相关库都能正常工作。

**Tip**

Java有很多可用的日志框架。 不要担心，如果上面的列表似乎令人困惑。 一般来说，你不需要改变你的日志依赖性，Spring Boot的默认设置就可以正常工作。

## Log 格式化

Spring Boot的默认日志输出如下所示：

:

:

:

:

2014-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] Initializing Spring embedded WebApplicationContext

2014-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader Root WebApplicationContext: initialization completed in 1358 ms

2014-03-05 10:57:51.698 INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean Mapping servlet: 'dispatcherServlet' to [/]

2014-03-05 10:57:51.702 INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean

Mapping filter: 'hiddenHttpMethodFilter' to: [/\*]

main] org.apache.catalina.core.StandardEngine :

2014-03-05 10:57:51.112 INFO 45469 --- [

Starting Servlet Engine: Apache Tomcat/7.0.52

以下项目被输出：

* 日期和时间 - 精确到毫秒，易于排序。
* 日志级别 - 错误，警告，信息，调试或跟踪。
* Process ID.
* A ---分隔符来区分实际日志消息的开始。
* 线程名称 - 括在方括号中（可能会截断控制台输出）。
* Logger name — 这通常是源类名称（通常缩写）。
* log message.

**Note**

Logback没有FATAL级别（它被映射到ERROR）

## Console 输出

默认日志配置会在写入消息时将消息回送到控制台。 默认情况下会记录ERROR，WARN和INFO级别的消息。 您也可以通过使用--debug标志启动应用程序来启用“调试”模式。

$ java -jar myapp.jar --debug

**Note**

你也可以在你的application.properties中指定debug = true。

当启用调试模式时，会选择一些核心loggers（嵌入式容器，Hibernate和Spring Boot）来输出更多的信息。 启用调试模式不会将您的应用程序配置为使用DEBUG级别记录所有消息。

或者，您可以通过使用trace标志（或application.properties中的trace = true）启动应用程序来启用“跟踪”模式。 这将启用对核心记录器（嵌入式容器，Hibernate模式生成和整个Spring产品组合）选择的跟踪记录。

### Color-coded 输出

如果您的终端支持ANSI，则会使用彩色输出来提高可读性。 你可以设置spring.output.ansi.enabled为

[supported value](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/ansi/AnsiOutput.Enabled.html) 来覆盖自动检测。

颜色编码使用％clr转换字进行配置。 在最简单的形式中，转换器将根据日志级别为输出着色，例如：

%clr(%5p)

The mapping of log level to a color is as follows:

|  |  |
| --- | --- |
| **Level** | **Color** |
| FATAL | Red |
| ERROR | Red |
| WARN  INFO | Yellow  Green |
| DEBUG | Green |
| TRACE | Green |

或者，您可以通过提供转换选项来指定应使用的颜色或样式。 例如，要使文本变成黄色：

%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}

支持以下颜色和样式：

* blue
* cyan
* faint
* green
* magenta
* red
* yellow

## File output

默认情况下，Spring Boot将只能输出到控制台，不会写入日志文件。 如果除了控制台输出之外还想写日志文件，则需要设置logging.file或logging.path属性（例如，在application.properties中）。

下表显示了logging。\*属性如何一起使用：

*Table 26.1. Logging properties*

|  |  |  |
| --- | --- | --- |
| **logging.filleogging.path**  *(none) (none)* | **Example** | **Description**  Console only logging. |
| Specific file *(none)* | my.log | Writes to the specified log file. Names can be an exact location or relative to the current directory. |
| *(none)* Specific directory | /var/log | Writes spring.log to the specified directory. Names can be an exact location or relative to the current directory. |

日志文件在达到10 MB时会循环，并且与控制台输出一样，默认情况下会记录ERROR，WARN和INFO级别的消息。

**Note**

日志记录系统在应用程序生命周期的早期初始化，因此在通过@PropertySource注解加载的属性文件中不会找到这样的日志记录属性。

**Tip**

日志记录属性独立于实际的日志记录基础结构。 因此，特定的配置keys（例如Logback的logback.configurationFile）不受Spring Boot的管理。

## Log 级别

所有支持的日志记录系统都可以使用'logging.level.\* = LEVEL'在Spring Environment中设置日志级别（例如在application.properties中），其中LEVEL是TRACE，DEBUG，INFO，WARN，ERROR， FATAL，OFF之一 。 根记录器可以使用logging.level.root进行配置。 示例application.properties：

**logging.level.root**=WARN **logging.level.org.springframework.web**=DEBUG **logging.level.org.hibernate**=ERROR

**Note**

默认情况下，Spring Boot会重新映射Thymeleaf INFO消息，以便在DEBUG级别进行记录。 这有助于减少标准日志输出中的noise。 有关如何在自己的配置中应用重新映射的详细信息，请参阅[LevelRemappingAppender](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/logging/logback/LevelRemappingAppender.java)。

## 自定义日志配置

可以通过在类路径中包含适当的库来激活各种日志记录系统，并且可以通过在类路径的根目录中或在Spring环境属性logging.config中指定的位置提供合适的配置文件来进一步进行定制。

您可以使用org.springframework.boot.logging.LoggingSystem系统属性强制Spring Boot使用特定的日志记录系统。 该值应该是LoggingSystem实现的完全限定(fully-qualified)的类名称。 您还可以完全禁用Spring Boot的日志记录配置，使用值为none。

**Note**

由于在创建ApplicationContext之前对日志进行初始化，因此无法在Spring @Configuration文件中控制来自@PropertySources的日志记录。 系统属性和传统的Spring Boot外部配置文件工作得很好。）

根据您的日志记录系统，将加载以下文件：

|  |  |
| --- | --- |
| **Logging System** | **Customization** |
| Logback | logback-spring.xml, logback- spring.groovy, logback.xml or logback.groovy |
| Log4j2  JDK (Java Util Logging) | log4j2-spring.xml or log4j2.xml  logging.properties |

**Note**

如果可能，我们建议您使用-spring变体进行日志记录配置（例如，logback-spring.xml而不是logback.xml）。 如果使用标准配置位置，Spring将无法完全控制日志初始化。

**Warning**

Java Util Logging存在已知的类加载问题，从“executable jar”运行时会导致问题。 我们建议您尽可能避免。

为了帮助定制，一些其他的属性从Spring环境中传递

系统属性：

|  |  |
| --- | --- |
| **Spring Environment**  logging.exception- conversion-word | **System Property Comments**  LOG\_EXCEPTION\_CONVERSION\_WTOhReDconversion word that’s  used when logging exceptions. |
| logging.file | LOG\_FILE Used in default log configuration if defined. |

|  |  |  |
| --- | --- | --- |
| **Spring Environment** | **System Property** | **Comments** |
| logging.path  logging.pattern.console | LOG\_PATH  CONSOLE\_LOG\_PATTERN | Used in default log configuration if defined.  The log pattern to use on the console (stdout). (Only supported with the default logback setup.) |
| logging.pattern.file  logging.pattern.level | FILE\_LOG\_PATTERN  LOG\_LEVEL\_PATTERN | The log pattern to use in a file (if LOG\_FILE enabled). (Only supported with the default logback setup.)  The format to use to render the log level (default %5p).  (Only supported with the default  logback setup.) |
| PID | PID | The current process ID (discovered if possible and when not already defined as an OS environment variable). |

所有支持的日志系统在解析其配置文件时都可以查询系统属性。 有关示例，请参阅spring-boot.jar中的默认配置。

**Tip**

如果您想在日志记录属性中使用占位符，则应该使用Spring Boot的语法，而不是基础框架的语法。 值得注意的是，如果您使用的是Logback，则应该使用：作为属性名称和其默认值之间的分隔符，而不是： -

**Tip**

您可以通过仅覆盖LOG\_LEVEL\_PATTERN（或使用Logback的logging.pattern.level）来向日志行添加MDC和其他临时内容。 例如，如果使用logging.pattern.level = user：％X {user}％5p，则默认日志格式将包含“user”的MDC条目（如果存在）。

2015-09-30 12:30:04.031 user:juergen INFO 22174 --- [ nio-8080-exec-0] demo.Controller Handling authenticated request

## Logback 扩展

Spring Boot包含许多对Logback的扩展，可以帮助进行高级配置。 您可以在您的logback-spring.xml配置文件中使用这些扩展。

**Note**

您不能在标准logback.xml配置文件中使用扩展，因为它太早加载。 您需要使用logback-spring.xml或定义logging.config属性。

**Warning**

这些扩展名不能用于Logback的配置扫描([configuration scanning](http://logback.qos.ch/manual/configuration.html#autoScan))。 如果您尝试这样做，则更改配置文件将导致类似于以下记录之一的错误：

ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProperty], current ElementPath is [[configuration][springProperty]]

ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile], current ElementPath is [[configuration][springProfile]]

### Profile-specific configuration

<springProfile>标签允许您根据活动的Spring配置文件选择性地包含或排除配置的各个部分。 配置文件部分在<configuration>元素的任何地方都受支持。 使用name属性来指定哪个配置文件接受配置。 可以使用逗号分隔列表指定多个配置文件。

**<springProfile name**=**"staging">**

*<!-- configuration to be enabled when the "staging" profile is active -->*

**</springProfile>**

**<springProfile name**=**"dev, staging">**

*<!-- configuration to be enabled when the "dev" or "staging" profiles are active -->*

**</springProfile>**

**<springProfile name**=**"!production">**

*<!-- configuration to be enabled when the "production" profile is not active -->*

**</springProfile>**

### Environment properties

<springProperty>标签允许您在Spring环境中使用Spring环境的属性。 如果要在您的logback配置中访问您的application.properties文件中的值，这可能很有用。 该标签的工作方式与Logback的标准<property>标签类似，但不是指定直接值，而是指定属性的来源（来自环境）。 如果需要将属性存储在本地范围以外的地方，可以使用scope属性。 如果您在环境中未设置属性的情况下需要回退值，则可以使用defaultValue属性。

**<springProperty scope**=**"context" name**=**"fluentHost" source**=**"myapp.fluentd.host" defaultValue**=**"localhost"/>**

**<appender name**=**"FLUENT" class**=**"ch.qos.logback.more.appenders.DataFluentAppender">**

**<remoteHost>**${fluentHost}**</remoteHost>**

...

**</appender>**

**Tip**

RelaxedPropertyResolver用于访问环境属性。 如果以虚线符号（my-property-name）指定源，则会尝试所有宽松的变体(relaxed variations)（myPropertyName，MY\_PROPERTY\_NAME等）。

# 开发 web 应用程序

Spring Boot非常适合Web应用程序的开发。 您可以使用嵌入式Tomcat，Jetty或Undertow轻松创建自包含的HTTP服务器。 大多数Web应用程序将使用spring-boot-starter-web模块来快速启动和运行。

如果您还没有开发Spring Boot Web应用程序，则可以按照“Hello World！”进行操作。 Getting started。

## ‘Spring Web MVC framework’

Spring Web MVC框架（通常简称为“Spring MVC”）是一个丰富的“模型视图控制器”Web框架。 Spring MVC允许您创建特殊的@Controller或@RestController bean来处理传入的HTTP请求。 您的控制器中的方法使用@RequestMapping注释映射到HTTP。

这里是一个典型的例子@RestController来提供JSON数据：

@RestController @RequestMapping(value="/users") **public class** MyRestController {

@RequestMapping(value="/{user}", method=RequestMethod.GET)

**public** User getUser(@PathVariable Long user) {

*// ...*

}

@RequestMapping(value="/{user}/customers", method=RequestMethod.GET) List<Customer> getUserCustomers(@PathVariable Long user) {

*// ...*

}

@RequestMapping(value="/{user}", method=RequestMethod.DELETE)

**public** User deleteUser(@PathVariable Long user) {

*// ...*

}

}

Spring MVC是核心Spring框架的一部分，详细信息见参考[reference](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle#mvc) [documentation](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle#mvc)。 在[spring.io/guides](http://spring.io/guides) 中也有几个指南，涵盖了Spring MVC。

### Spring MVC auto-configuration

Spring Boot为Spring MVC提供了自动配置，可以与大多数应用程序配合使用。 自动配置在Spring的默认设置之上添加以下功能：

* 包含ContentNegotiatingViewResolver和BeanNameViewResolver bean。
* 支持提供静态资源，包括对WebJars的支持（见下文）。
* 自动注册Converter，GenericConverter，Formatter bean。
* 支持HttpMessageConverters（见下文）。
* 自动注册MessageCodesResolver（见下文）。
* 静态index.html支持。
* 自定义Favicon支持（见下文）。
* 自动使用可配置的WebBindingInitializer bean（见下文）。

如果你想保持Spring Boot MVC的功能，并且你只是想添加额外的[MVC](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle#mvc) [configuration](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle#mvc) （拦截器，格式化器，视图控制器等），你可以添加你自己的@Configuration类型的WebMvcConfigurerAdapter，但是没有@EnableWebMvc。 如果您希望提供RequestMappingHandlerMapping，RequestMappingHandlerAdapter或ExceptionHandlerExceptionResolver的自定义实例，则可以声明提供此类组件的WebMvcRegistrationsAdapter实例。

如果你想完全控制Spring MVC，你可以添加你自己的用@EnableWebMvc注解的@Configuration。

### HttpMessageConverters

Spring MVC使用HttpMessageConverter接口来转换HTTP请求和响应。 明智的默认是包括在盒子，例如对象可以自动转换为JSON（使用Jackson）或XML（使用Jackson XML扩展如果可用，否则使用JAXB）。 字符串默认使用UTF-8编码。

如果你需要添加或定制转换器，你可以使用Spring Boot的HttpMessageConverters

类：

**import** org.springframework.boot.autoconfigure.web.HttpMessageConverters;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.http.converter.\*;

@Configuration

**public class** MyConfiguration {

@Bean

**public** HttpMessageConverters customConverters() { HttpMessageConverter<?> additional = ...

HttpMessageConverter<?> another = ...

**return new** HttpMessageConverters(additional, another);

}

}

存在于上下文中的任何HttpMessageConverter bean将被添加到转换器列表中。 您也可以通过这种方式覆盖默认转换器。

### 自定义JSON序列化器和反序列化器

如果您使用Jackson来序列化和反序列化JSON数据，则可能需要编写自己的JsonSerializer和JsonDeserializer类。 自定义序列化器通常通过[registered with](http://wiki.fasterxml.com/JacksonHowToCustomDeserializers) [Jackson via a Module](http://wiki.fasterxml.com/JacksonHowToCustomDeserializers)，但Spring Boot提供了一个替代的@JsonComponent注释，这使得更容易直接注册Spring Bean。

您可以直接在JsonSerializer或JsonDeserializer实现上使用@JsonComponent。 您也可以在包含序列化器/反序列化器的类中使用它作为内部类。 例如：

**import** java.io.\*;

**import** com.fasterxml.jackson.core.\*;

**import** com.fasterxml.jackson.databind.\*;

**import** org.springframework.boot.jackson.\*;

@JsonComponent

**public class** Example {

**public static class** Serializer **extends** JsonSerializer<SomeObject> {

*// ...*

}

**public static class** Deserializer **extends** JsonDeserializer<SomeObject> {

*// ...*

}

}

ApplicationContext中的所有@JsonComponent bean将自动注册到Jackson，并且由于@JsonComponent使用@Component进行元注释，所以通常的组件扫描规则适用。

Spring Boot还提供了[JsonObjectSerializer](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jackson/JsonObjectSerializer.java) 和[JsonObjectDeserializer](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jackson/JsonObjectDeserializer.java) 基类，在序列化对象时为标准Jackson版本提供了有用的替代方法。 有关详细信息，请参阅Javadoc。

### MessageCodesResolver

Spring MVC有一个策略来生成错误代码，用于从绑定错误中呈现错误消息：MessageCodesResolver。 如果您设置spring.mvc.message-codes-resolver.format属性PREFIX\_ERROR\_CODE或POSTFIX\_ERROR\_CODE（请参阅DefaultMessageCodesResolver.Format中的枚举），Spring Boot将为您创建一个。

### 静态内容

默认情况下，Spring Boot将从类路径中的/ static（或/ public或/ resources或/ META-INF / resources）目录或从ServletContext的根目录中提供静态内容。 它使用Spring MVC中的ResourceHttpRequestHandler，因此可以通过添加自己的WebMvcConfigurerAdapter并重写addResourceHandlers方法来修改该行为。

在stand-alone Web应用程序中，容器中的默认servlet也被启用，并作为后备，如果Spring决定不处理，则从ServletContext的根目录提供内容。 大多数情况下，这是不会发生的（除非你修改默认的MVC配置），因为Spring总是能够通过DispatcherServlet处理请求。

默认情况下，资源映射到/ \*\*，但可以通过spring.mvc.static-path- pattern调整。 例如，将所有资源重定位到/ resources / \*\*可以实现如下：

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

您也可以使用spring.resources.static-locations（使用目录位置列表替换缺省值）来自定义静态资源位置。 如果您这样做，默认的欢迎页面检测将切换到您的自定义位置。 因此，如果您的任何位置在启动时都有index.html，它将成为应用程序的主页。

除了上面的“标准”静态资源位置之外，Webjars内容还有一个特殊情况。 如果以Webjars格式打包，那么在/ webjars / \*\*中具有路径的任何资源都将从jar文件提供。

**Tip**

如果您的应用程序将被打包为jar，请不要使用src / main / webapp目录。 虽然这个目录是一个通用的标准，但是它只能用于war包装，如果你生成一个jar包，它会被大多数的构建工具默默地忽略。

Spring Boot还支持Spring MVC提供的高级资源处理功能，允许使用例如缓存清除静态资源或使用Webjars的版本不可知的URL。

要为Webjars使用版本不可知的URL，只需添加webjars-locator依赖项即可。 然后声明你的Webjar，例如jQuery，作为“/webjars/jquery/dist/jquery.min.js”，导致“/webjars/jquery/xyz/dist/jquery.min.js”其中xyz是Webjar版本。

**Note**

如果您使用的是JBoss，则需要声明webjars-locator-jboss-vfs依赖项，而不是webjars-locator; 否则所有的Webjar将解析为404。

要使用缓存清除，以下配置将为所有静态资源配置缓存清除解决方案，从而在URL中有效添加内容哈希，如<link href =“/ css / spring-2a2d595e6ed9a0b24f027f2b63b134d6.css”/>

**spring.resources.chain.strategy.content.enabled**=true **spring.resources.chain.strategy.content.paths**=/\*\*

**Note**

在模板中，资源链接在运行时被重写，这要归功于为Thymeleaf和FreeMarker自动配置的ResourceUrlEncodingFilter。 使用JSP时，您应该手动声明此过滤器。 其他模板引擎现在不会自动支持，但可以使用自定义模板宏/助手和[ResourceUrlProvider](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/web/servlet/resource/ResourceUrlProvider.html)的使用。

例如，在使用JavaScript模块加载程序动态加载资源时，重命名文件不是一个选项。 这就是为什么其他策略也支持并可以结合的原因。 “fixed”策略将在URL中添加静态版本字符串，而不更改文件名称：

**spring.resources.chain.strategy.content.enabled**=true **spring.resources.chain.strategy.content.paths**=/\*\* **spring.resources.chain.strategy.fixed.enabled**=true **spring.resources.chain.strategy.fixed.paths**=/js/lib/ **spring.resources.chain.strategy.fixed.version**=v12

使用此配置，位于“/ js / lib /”下的JavaScript模块将使用固定的版本控制策略“/v12/js/lib/mymodule.js”，而其他资源仍将使用内容<link href =“/ css /spring-2a2d595e6ed9a0b24f027f2b63b134d6.css“/>。

有关更多受支持的选项，请参阅[ResourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ResourceProperties.java) 。

**Tip**

这个特性已经在专门的[blog post](https://spring.io/blog/2014/07/24/spring-framework-4-1-handling-static-web-resources) 和Spring框架的参考文档([reference documentation](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#mvc-config-static-resources))中有详细的介绍。

### Custom Favicon

Spring Boot会在配置的静态内容位置和类路径的根目录（按此顺序）中查找favicon.ico。 如果这样的文件存在，它会自动用作应用程序的图标。

### ConfigurableWebBindingInitializer

Spring MVC uses a WebBindingInitializer to initialize a WebDataBinder for a particular request. If you create your own ConfigurableWebBindingInitializer @Bean, Spring Boot will automatically configure Spring MVC to use it.

### Template engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies including Thymeleaf, FreeMarker and JSPs. Many other templating engines also ship their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

* [FreeMarker](http://freemarker.org/docs/)
* [Groovy](http://docs.groovy-lang.org/docs/next/html/documentation/template-engines.html#_the_markuptemplateengine)
* [Thymeleaf](http://www.thymeleaf.org/)
* [Mustache](http://mustache.github.io/)

**Tip**

JSPs should be avoided if possible, there are several [known limitations](#_bookmark155) when using them with embedded servlet containers.

When you’re using one of these templating engines with the default configuration, your templates will be picked up automatically from src/main/resources/templates.

**Tip**

IntelliJ IDEA orders the classpath differently depending on how you run your application. Running your application in the IDE via its main method will result in a different ordering to when you run your application using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the templates on the classpath. If you’re affected by this problem you can reorder the classpath in the IDE to place the module’s classes and resources first. Alternatively, you can configure the template prefix to search every templates directory on the classpath: classpath\*:/templates/.

### Error Handling

Spring Boot provides an /error mapping by default that handles all errors in a sensible way, and it is registered as a ‘global’ error page in the servlet container. For machine clients it will produce a JSON response with details of the error, the HTTP status and the exception message. For browser clients there is a ‘whitelabel’ error view that renders the same data in HTML format (to customize it just add a View that resolves to ‘error’). To replace the default behaviour completely you can implement ErrorController and register a bean definition of that type, or simply add a bean of type ErrorAttributes to use the existing mechanism but replace the contents.

**Tip**

The BasicErrorController can be used as a base class for a custom ErrorController. This is particularly useful if you want to add a handler for a new content type (the default is to handle text/html specifically and provide a fallback for everything else). To do that just extend BasicErrorController and add a public method with a @RequestMapping that has a produces attribute, and create a bean of your new type.

You can also define a @ControllerAdvice to customize the JSON document to return for a particular controller and/or exception type.

@ControllerAdvice(basePackageClasses = FooController.class)

**public class** FooControllerAdvice **extends** ResponseEntityExceptionHandler {

@ExceptionHandler(YourException.class) @ResponseBody

ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) { HttpStatus status = getStatus(request);

**return new** ResponseEntity<>(**new** CustomErrorType(status.value(), ex.getMessage()), status);

}

**private** HttpStatus getStatus(HttpServletRequest request) {

Integer statusCode = (Integer) request.getAttribute(***"javax.servlet.error.status\_code"***);

**if** (statusCode == null) {

**return** HttpStatus.INTERNAL\_SERVER\_ERROR;

}

**return** HttpStatus.valueOf(statusCode);

}

}

In the example above, if YourException is thrown by a controller defined in the same package as FooController, a json representation of the CustomErrorType POJO will be used instead of the ErrorAttributes representation.

#### Custom error pages

If you want to display a custom HTML error page for a given status code, you add a file to an /error folder. Error pages can either be static HTML (i.e. added under any of the static resource folders) or built using templates. The name of the file should be the exact status code or a series mask.

For example, to map 404 to a static HTML file, your folder structure would look like this:

src/

+- main/

+- java/

| + <source code>

+- resources/

+- public/

+- error/

| +- 404.html

+- <other public assets>

To map all 5xx errors using a FreeMarker template, you’d have a structure like this:

src/

+- main/

+- java/

| + <source code>

+- resources/

+- templates/

+- error/

| +- 5xx.ftl

+- <other templates>

For more complex mappings you can also add beans that implement the ErrorViewResolver

interface.

**public class** MyErrorViewResolver **implements** ErrorViewResolver {

@Override

**public** ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, Object> model) {

*// Use the request or status to optionally return a ModelAndView*

**return** ...

}

}

You can also use regular Spring MVC features like [@ExceptionHandler methods](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#mvc-exceptionhandlers) and

[@ControllerAdvice](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#mvc-ann-controller-advice). The ErrorController will then pick up any unhandled exceptions.

#### Mapping error pages outside of Spring MVC

For applications that aren’t using Spring MVC, you can use the ErrorPageRegistrar interface to directly register ErrorPages. This abstraction works directly with the underlying embedded servlet container and will work even if you don’t have a Spring MVC DispatcherServlet.

@Bean

**public** ErrorPageRegistrar errorPageRegistrar(){

**return new** MyErrorPageRegistrar();

}

*// ...*

**private static class** MyErrorPageRegistrar **implements** ErrorPageRegistrar { @Override

**public void** registerErrorPages(ErrorPageRegistry registry) { registry.addErrorPages(**new** ErrorPage(HttpStatus.BAD\_REQUEST, ***"/400"***));

}

}

N.B. if you register an ErrorPage with a path that will end up being handled by a Filter (e.g. as is common with some non-Spring web frameworks, like Jersey and Wicket), then the Filter has to be explicitly registered as an ERROR dispatcher, e.g.

@Bean

**public** FilterRegistrationBean myFilter() {

FilterRegistrationBean registration = **new** FilterRegistrationBean(); registration.setFilter(**new** MyFilter());

...

registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.**class**));

**return** registration;

}

(the default FilterRegistrationBean does not include the ERROR dispatcher type).

#### Error Handling on WebSphere Application Server

When deployed to a servlet container, a Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet’s service method. You should disable this behaviour by setting com.ibm.ws.webcontainer.invokeFlushAfterService to false

### Spring HATEOAS

If you’re developing a RESTful API that makes use of hypermedia, Spring Boot provides auto- configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use @EnableHypermediaSupport and registers a number of beans to ease building hypermedia-based applications including a LinkDiscoverers (for client side support) and an ObjectMapper configured to correctly marshal responses into the desired representation. The ObjectMapper will be customized based on the spring.jackson.\* properties or a Jackson2ObjectMapperBuilder bean if one exists.

You can take control of Spring HATEOAS’s configuration by using @EnableHypermediaSupport. Note that this will disable the ObjectMapper customization described above.

### CORS support

[Cross-origin resource sharing](http://en.wikipedia.org/wiki/Cross-origin_resource_sharing) (CORS) is a [W3C specification](http://www.w3.org/TR/cors/) implemented by [most browsers](http://caniuse.com/#feat%3Dcors) that allows you to specify in a flexible way what kind of cross domain requests are authorized, instead of using some less secure and less powerful approaches like IFRAME or JSONP.

As of version 4.2, Spring MVC [supports CORS](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#cors) out of the box. Using [controller method CORS](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#_controller_method_cors_configuration) [configuration](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#_controller_method_cors_configuration) with [@CrossOrigin](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html) annotations in your Spring Boot application does not require any specific configuration. [Global CORS configuration](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#_global_cors_configuration) can be defined by registering a WebMvcConfigurer bean with a customized addCorsMappings(CorsRegistry) method:

@Configuration

**public class** MyConfiguration {

@Bean

**public** WebMvcConfigurer corsConfigurer() {

**return new** WebMvcConfigurerAdapter() { @Override

**public void** addCorsMappings(CorsRegistry registry) { registry.addMapping(***"/api/\*\*"***);

}

};

}

}

## JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints you can use one of the available implementations instead of Spring MVC. Jersey 1.x and Apache CXF work quite well out of the box if you just register their Servlet or Filter as a @Bean in your application context. Jersey 2.x has some native Spring support so we also provide auto-configuration support for it in Spring Boot together with a starter.

To get started with Jersey 2.x just include the spring-boot-starter-jersey as a dependency and then you need one @Bean of type ResourceConfig in which you register all the endpoints:

@Component

**public class** JerseyConfig **extends** ResourceConfig {

**public** JerseyConfig() { register(Endpoint.**class**);

}

}

**Warning**

Jersey’s support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in WEB-INF/classes when running an executable war file. To avoid this limitation, the packages method should not be used and endpoints should be registered individually using the register method as shown above.

You can also register an arbitrary number of beans implementing ResourceConfigCustomizer for more advanced customizations.

All the registered endpoints should be @Components with HTTP resource annotations (@GET etc.), e.g.

@Component @Path("/hello")

**public class** Endpoint {

@GET

**public** String message() {

**return *"Hello"***;

}

}

Since the Endpoint is a Spring @Component its lifecycle is managed by Spring and you can @Autowired dependencies and inject external configuration with @Value. The Jersey servlet will be registered and mapped to /\* by default. You can change the mapping by adding @ApplicationPath to your ResourceConfig.

By default Jersey will be set up as a Servlet in a @Bean of type ServletRegistrationBean named jerseyServletRegistration. By default, the servlet will be initialized lazily but you can customize it with spring.jersey.servlet.load-on-startup .You can disable or override that bean by creating one of your own with the same name. You can also use a Filter instead of a Servlet by setting spring.jersey.type=filter (in which case the @Bean to replace or override is jerseyFilterRegistration). The servlet has an @Order which you can set with spring.jersey.filter.order. Both the Servlet and the Filter registrations can be given init parameters using spring.jersey.init.\* to specify a map of properties.

There is a [Jersey sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-jersey) so you can see how to set things up. There is also a [Jersey 1.x sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-jersey1). Note that in the Jersey 1.x sample that the spring-boot maven plugin has been configured to unpack some Jersey jars so they can be scanned by the JAX-RS implementation (because the sample asks for them to be scanned in its Filter registration). You may need to do the same if any of your JAX- RS resources are packaged as nested jars.

## Embedded servlet container support

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers will simply use the appropriate ‘Starter’ to obtain a fully configured instance. By default the embedded server will listen for HTTP requests on port 8080.

**Warning**

If you choose to use Tomcat on CentOS be aware that, by default, a temporary directory is used to store compiled JSPs, file uploads etc. This directory may be deleted by tmpwatch while your application is running leading to failures. To avoid this, you may want to customize

your tmpwatch configuration so that tomcat.\* directories are not deleted, or configure

server.tomcat.basedir so that embedded Tomcat uses a different location.

### Servlets, Filters, and listeners

When using an embedded servlet container you can register Servlets, Filters and all the listeners from the Servlet spec (e.g. HttpSessionListener) either by using Spring beans or by scanning for Servlet components.

#### Registering Servlets, Filters, and listeners as Spring beans

Any Servlet, Filter or Servlet \*Listener instance that is a Spring bean will be registered with the embedded container. This can be particularly convenient if you want to refer to a value from your application.properties during configuration.

By default, if the context contains only a single Servlet it will be mapped to /. In the case of multiple Servlet beans the bean name will be used as a path prefix. Filters will map to /\*.

If convention-based mapping is not flexible enough you can use the ServletRegistrationBean, FilterRegistrationBean and ServletListenerRegistrationBean classes for complete control.

### Servlet Context Initialization

Embedded servlet containers will not directly execute the Servlet 3.0+ javax.servlet.ServletContainerInitializer interface, or Spring’s org.springframework.web.WebApplicationInitializer interface. This is an intentional design decision intended to reduce the risk that 3rd party libraries designed to run inside a war will break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the org.springframework.boot.context.embedded.ServletContextInitializer interface. The single onStartup method provides access to the ServletContext, and can easily be used as an adapter to an existing WebApplicationInitializer if necessary.

#### Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of @WebServlet, @WebFilter, and

@WebListener annotated classes can be enabled using @ServletComponentScan.

**Tip**

@ServletComponentScan will have no effect in a standalone container, where the container’s built-in discovery mechanisms will be used instead.

### The EmbeddedWebApplicationContext

Under the hood Spring Boot uses a new type of ApplicationContext for embedded servlet container support. The EmbeddedWebApplicationContext is a special type of WebApplicationContext that bootstraps itself by searching for a single EmbeddedServletContainerFactory bean. Usually a TomcatEmbeddedServletContainerFactory, JettyEmbeddedServletContainerFactory, or UndertowEmbeddedServletContainerFactory will have been auto-configured.

**Note**

You usually won’t need to be aware of these implementation classes. Most applications will be auto-configured and the appropriate ApplicationContext and EmbeddedServletContainerFactory will be created on your behalf.

### Customizing embedded servlet containers

Common servlet container settings can be configured using Spring Environment properties. Usually you would define the properties in your application.properties file.

Common server settings include:

* Network settings: listen port for incoming HTTP requests (server.port), interface address to bind to server.address, etc.
* Session settings: whether the session is persistent (server.session.persistence), session timeout (server.session.timeout), location of session data (server.session.store-dir) and session-cookie configuration (server.session.cookie.\*).
* Error management: location of the error page (server.error.path), etc.
* [SSL](#_bookmark464)
* [HTTP compression](#_bookmark486)

Spring Boot tries as much as possible to expose common settings but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see server.tomcat and server.undertow). For instance, [access logs](#_bookmark465) can be configured with specific features of the embedded servlet container.

**Tip**

See the [ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java) class for a complete list.

#### Programmatic customization

If you need to configure your embedded servlet container programmatically you can register a Spring bean that implements the EmbeddedServletContainerCustomizer interface. EmbeddedServletContainerCustomizer provides access to the ConfigurableEmbeddedServletContainer which includes numerous customization setter methods.

**import** org.springframework.boot.context.embedded.\*;

**import** org.springframework.stereotype.Component;

@Component

**public class** CustomizationBean **implements** EmbeddedServletContainerCustomizer {

@Override

**public void** customize(ConfigurableEmbeddedServletContainer container) { container.setPort(9000);

}

}

#### Customizing ConfigurableEmbeddedServletContainer directly

If the above customization techniques are too limited, you can register the TomcatEmbeddedServletContainerFactory, JettyEmbeddedServletContainerFactory or UndertowEmbeddedServletContainerFactory bean yourself.

@Bean

**public** EmbeddedServletContainerFactory servletContainer() { TomcatEmbeddedServletContainerFactory factory = **new** TomcatEmbeddedServletContainerFactory(); factory.setPort(9000);

factory.setSessionTimeout(10, TimeUnit.MINUTES);

factory.addErrorPages(**new** ErrorPage(HttpStatus.NOT\_FOUND, ***"/notfound.html"***));

**return** factory;

}

Setters are provided for many configuration options. Several protected method ‘hooks’ are also provided should you need to do something more exotic. See the source code documentation for details.

### JSP limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

* With Tomcat it should work if you use war packaging, i.e. an executable war will work, and will also be deployable to a standard container (not limited to, but including Tomcat). An executable jar will not work because of a hard coded file pattern in Tomcat.
* With Jetty it should work if you use war packaging, i.e. an executable war will work, and will also be deployable to any standard container.
* Undertow does not support JSPs.
* Creating a custom error.jsp page won’t override the default view for [error handling](#_bookmark139), [custom error pages](#_bookmark140) should be used instead.

There is a [JSP sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-web-jsp) so you can see how to set things up.

# Security

If Spring Security is on the classpath then web applications will be secure by default with ‘basic’ authentication on all HTTP endpoints. To add method-level security to a web application you can also add @EnableGlobalMethodSecurity with your desired settings. Additional information can be found in the [Spring Security Reference](http://docs.spring.io/spring-security/site/docs/4.2.3.RELEASE/reference/htmlsingle#jc-method).

The default AuthenticationManager has a single user (‘user’ username and random password, printed at INFO level when the application starts up)

Using default security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35

**Note**

If you fine-tune your logging configuration, ensure that the org.springframework.boot.autoconfigure.security category is set to log INFO messages, otherwise the default password will not be printed.

You can change the password by providing a security.user.password. This and other useful properties are externalized via [SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java) (properties prefix "security").

The default security configuration is implemented in SecurityAutoConfiguration and in the classes imported from there (SpringBootWebSecurityConfiguration for web security and AuthenticationManagerConfiguration for authentication configuration which is also relevant in non-web applications). To switch off the default web application security configuration completely you can add a bean with @EnableWebSecurity (this does not disable the authentication manager configuration or Actuator’s security). To customize it you normally use external properties and beans of type WebSecurityConfigurerAdapter (e.g. to add form-based login).

**Note**

If you add @EnableWebSecurity and also disable Actuator security, you will get the default form-based login for the entire application unless you add a custom WebSecurityConfigurerAdapter.

To also switch off the authentication manager configuration you can add a bean of type AuthenticationManager, or else configure the global AuthenticationManager by autowiring an AuthenticationManagerBuilder into a method in one of your @Configuration classes. There are several secure applications in the [Spring Boot samples](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/) to get you started with common use cases.

The basic features you get out of the box in a web application are:

* An AuthenticationManager bean with in-memory store and a single user (see

SecurityProperties.User for the properties of the user).

* Ignored (insecure) paths for common static resource locations (/css/\*\*, /js/\*\*, /images/\*\*, / webjars/\*\* and \*\*/favicon.ico).
* HTTP Basic security for all other endpoints.
* Security events published to Spring’s ApplicationEventPublisher (successful and unsuccessful authentication and access denied).
* Common low-level features (HSTS, XSS, CSRF, caching) provided by Spring Security are on by default.

All of the above can be switched on and off or modified using external properties (security.\*). To override the access rules without changing any other auto-configured features add a @Bean of type WebSecurityConfigurerAdapter with @Order(SecurityProperties.ACCESS\_OVERRIDE\_ORDER) and configure it to meet your needs.

**Note**

By default, a WebSecurityConfigurerAdapter will match any path. If you don’t want to completely override Spring Boot’s auto-configured access rules, your adapter must explicitly configure the paths that you do want to override.

## OAuth2

If you have spring-security-oauth2 on your classpath you can take advantage of some auto- configuration to make it easy to set up Authorization or Resource Server. For full details, see the [Spring](http://projects.spring.io/spring-security-oauth/docs/oauth2.html) [Security OAuth 2 Developers Guide](http://projects.spring.io/spring-security-oauth/docs/oauth2.html).

### Authorization Server

To create an Authorization Server and grant access tokens you need to use @EnableAuthorizationServer and provide security.oauth2.client.client-id and security.oauth2.client.client-secret] properties. The client will be registered for you in an in-memory repository.

Having done that you will be able to use the client credentials to create an access token, for example:

$ curl client:secret@localhost:8080/oauth/token -d grant\_type=password -d username=user -d password=pwd

The basic auth credentials for the /token endpoint are the client-id and client-secret. The user credentials are the normal Spring Security user details (which default in Spring Boot to “user” and a random password).

To switch off the auto-configuration and configure the Authorization Server features yourself just add a

@Bean of type AuthorizationServerConfigurer.

### Resource Server

To use the access token you need a Resource Server (which can be the same as the Authorization Server). Creating a Resource Server is easy, just add @EnableResourceServer and provide some configuration to allow the server to decode access tokens. If your application is also an Authorization Server it already knows how to decode tokens, so there is nothing else to do. If your app is a standalone service then you need to give it some more configuration, one of the following options:

* security.oauth2.resource.user-info-uri to use the /me resource (e.g. https:// uaa.run.pivotal.io/userinfo on Pivotal Web Services (PWS))
* security.oauth2.resource.token-info-uri to use the token decoding endpoint (e.g.

https://uaa.run.pivotal.io/check\_token on PWS).

If you specify both the user-info-uri and the token-info-uri then you can set a flag to say that one is preferred over the other (prefer-token-info=true is the default).

Alternatively (instead of user-info-uri or token-info-uri) if the tokens are JWTs you can configure a security.oauth2.resource.jwt.key-value to decode them locally (where the key is a verification key). The verification key value is either a symmetric secret or PEM-encoded RSA public key. If you don’t have the key and it’s public you can provide a URI where it can be downloaded (as a JSON object with a “value” field) with security.oauth2.resource.jwt.key-uri. E.g. on PWS:

$ curl https://uaa.run.pivotal.io/token\_key

{"alg":"SHA256withRSA","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUBLIC KEY-----\n"}

Additionally, if your authorization server has an endpoint that returns a set of JSON Web Keys(JWKs), you can configure security.oauth2.resource.jwk.key-set-uri. E.g. on PWS:

$ curl https://uaa.run.pivotal.io/token\_keys

{"keys":[{"kid":"key-1","alg":"RS256","value":"-----BEGIN PUBLIC KEY-----\nMIIBI...\n-----END PUBLIC KEY-----\n"]}

**Note**

Configuring both JWT and JWK properties will cause an error. Only one of security.oauth2.resource.jwt.key-uri (or security.oauth2.resource.jwt.key-value) and security.oauth2.resource.jwk.key-set-uri should be configured.

**Warning**

If you use the security.oauth2.resource.jwt.key-uri or

`security.oauth2.resource.jwk.key-set-uri, ` the authorization server needs to be running when your application starts up. It will log a warning if it can’t find the key, and tell you what to do to fix it.

OAuth2 resources are protected by a filter chain with order security.oauth2.resource.filter- order and the default is after the filter protecting the actuator endpoints by default (so actuator endpoints will stay on HTTP Basic unless you change the order).

## Token Type in User Info

Google, and certain other 3rd party identity providers, are more strict about the token type name that is sent in the headers to the user info endpoint. The default is “Bearer” which suits most providers and matches the spec, but if you need to change it you can set security.oauth2.resource.token- type.

## Customizing the User Info RestTemplate

If you have a user-info-uri, the resource server features use an OAuth2RestTemplate internally to fetch user details for authentication. This is provided as a @Bean of type UserInfoRestTemplateFactory. The default should be fine for most providers, but occasionally you might need to add additional interceptors, or change the request authenticator (which is how the token gets attached to outgoing requests). To add a customization just create a bean of type UserInfoRestTemplateCustomizer - it has a single method that will be called after the bean is created but before it is initialized. The rest template that is being customized

here is *only* used internally to carry out authentication. Alternatively, you could define your own

UserInfoRestTemplateFactory @Bean to take full control.

**Tip**

To set an RSA key value in YAML use the “pipe” continuation marker to split it over multiple lines (“|”) and remember to indent the key value (it’s a standard YAML language feature). Example:

**security**:

**oauth2**:

**resource**:

**jwt**:

**keyValue**: |

-----BEGIN PUBLIC KEY----- MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC...

-----END PUBLIC KEY-----

### Client

To make your web-app into an OAuth2 client you can simply add @EnableOAuth2Client and Spring Boot will create a OAuth2ClientContext and OAuth2ProtectedResourceDetails that are necessary to create an OAuth2RestOperations. Spring Boot does not automatically create such bean but you can easily create your own:

@Bean

**public** OAuth2RestTemplate oauth2RestTemplate(OAuth2ClientContext oauth2ClientContext, OAuth2ProtectedResourceDetails details) {

**return new** OAuth2RestTemplate(details, oauth2ClientContext);

}

**Note**

You may want to add a qualifier and review your configuration as more than one RestTemplate

may be defined in your application.

This configuration uses security.oauth2.client.\* as credentials (the same as you might be using in the Authorization Server), but in addition it will need to know the authorization and token URIs in the Authorization Server. For example:

**application.yml.**

**security**:

**oauth2**:

**client**:

**clientId**: bd1c0a783ccdd1c9b9e4

**clientSecret**: 1a9030fbca47a5b2c28e92f19050bb77824b5ad1 **accessTokenUri**: https://github.com/login/oauth/access\_token **userAuthorizationUri**: https://github.com/login/oauth/authorize **clientAuthenticationScheme**: form

An application with this configuration will redirect to Github for authorization when you attempt to use the OAuth2RestTemplate. If you are already signed into Github you won’t even notice that it has authenticated. These specific credentials will only work if your application is running on port 8080 (register your own client app in Github or other provider for more flexibility).

To limit the scope that the client asks for when it obtains an access token you can set

security.oauth2.client.scope (comma separated or an array in YAML). By default the scope

is empty and it is up to Authorization Server to decide what the defaults should be, usually depending on the settings in the client registration that it holds.

**Note**

There is also a setting for security.oauth2.client.client-authentication-scheme which defaults to “header” (but you might need to set it to “form” if, like Github for instance, your OAuth2 provider doesn’t like header authentication). In fact, the security.oauth2.client.\* properties are bound to an instance of AuthorizationCodeResourceDetails so all its properties can be specified.

**Tip**

In a non-web application you can still create an OAuth2RestOperations and it is still wired into the security.oauth2.client.\* configuration. In this case it is a “client credentials token grant” you will be asking for if you use it (and there is no need to use @EnableOAuth2Client or @EnableOAuth2Sso). To prevent that infrastructure to be defined, just remove the security.oauth2.client.client-id from your configuration (or make it the empty string).

### Single Sign On

An OAuth2 Client can be used to fetch user details from the provider (if such features are available) and then convert them into an Authentication token for Spring Security. The Resource Server above support this via the user-info-uri property This is the basis for a Single Sign On (SSO) protocol based on OAuth2, and Spring Boot makes it easy to participate by providing an annotation @EnableOAuth2Sso. The Github client above can protect all its resources and authenticate using the Github /user/ endpoint, by adding that annotation and declaring where to find the endpoint (in addition to the security.oauth2.client.\* configuration already listed above):

**application.yml.**

**security**:

**oauth2**:

...

**resource**:

**userInfoUri**: https://api.github.com/user

**preferTokenInfo**: **false**

Since all paths are secure by default, there is no “home” page that you can show to unauthenticated users and invite them to login (by visiting the /login path, or the path specified by security.oauth2.sso.login-path).

To customize the access rules or paths to protect, so you can add a “home” page for instance, @EnableOAuth2Sso can be added to a WebSecurityConfigurerAdapter and the annotation will cause it to be decorated and enhanced with the necessary pieces to get the /login path working. For example, here we simply allow unauthenticated access to the home page at "/" and keep the default for everything else:

@Configuration

**static class** WebSecurityConfiguration **extends** WebSecurityConfigurerAdapter {

@Override

**public void** init(WebSecurity web) { web.ignoring().antMatchers(***"/"***);

}

@Override

**protected void** configure(HttpSecurity http) **throws** Exception { http.antMatcher(***"/\*\*"***).authorizeRequests().anyRequest().authenticated();

}

}

## Actuator Security

If the Actuator is also in use, you will find:

* The management endpoints are secure even if the application endpoints are insecure.
* Security events are transformed into AuditEvent instances and published to the

AuditEventRepository.

* The default user will have the ACTUATOR role as well as the USER role.

The Actuator security features can be modified using external properties (management.security.\*). To override the application access rules add a @Bean of type WebSecurityConfigurerAdapter and use @Order(SecurityProperties.ACCESS\_OVERRIDE\_ORDER) if you *don’t* want to override the actuator access rules, or @Order(ManagementServerProperties.ACCESS\_OVERRIDE\_ORDER) if you *do* want to override the actuator access rules.

# Working with SQL databases

The Spring Framework provides extensive support for working with SQL databases. From direct JDBC access using JdbcTemplate to complete ‘object relational mapping’ technologies such as Hibernate. Spring Data provides an additional level of functionality, creating Repository implementations directly from interfaces and using conventions to generate queries from your method names.

## Configure a DataSource

Java’s javax.sql.DataSource interface provides a standard method of working with database connections. Traditionally a DataSource uses a URL along with some credentials to establish a database connection.

**Tip**

Check also [the ‘How-to’ section](#_bookmark505) for more advanced examples, typically to take full control over the configuration of the DataSource.

### Embedded Database Support

It’s often convenient to develop applications using an in-memory embedded database. Obviously, in- memory databases do not provide persistent storage; you will need to populate your database when your application starts and be prepared to throw away data when your application ends.

**Tip**

The ‘How-to’ section includes a [*section on how to initialize a database*](#_bookmark517)

Spring Boot can auto-configure embedded [H2](http://www.h2database.com/), [HSQL](http://hsqldb.org/) and [Derby](http://db.apache.org/derby/) databases. You don’t need to provide any connection URLs, simply include a build dependency to the embedded database that you want to use.

**Note**

If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set spring.datasource.generate-unique-name to true.

For example, typical POM dependencies would be:

**<dependency>**

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

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

**</dependency>**

**<dependency>**

**<groupId>**org.hsqldb**</groupId>**

**<artifactId>**hsqldb**</artifactId>**

**<scope>**runtime**</scope>**

**</dependency>**

**Note**

You need a dependency on spring-jdbc for an embedded database to be auto-configured. In this example it’s pulled in transitively via spring-boot-starter-data-jpa.

**Tip**

If, for whatever reason, you do configure the connection URL for an embedded database, care should be taken to ensure that the database’s automatic shutdown is disabled. If you’re using H2 you should use DB\_CLOSE\_ON\_EXIT=FALSE to do so. If you’re using HSQLDB, you should ensure that shutdown=true is not used. Disabling the database’s automatic shutdown allows Spring Boot to control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

### Connection to a production database

Production database connections can also be auto-configured using a pooling DataSource. Here’s the algorithm for choosing a specific implementation:

* We prefer the Tomcat pooling DataSource for its performance and concurrency, so if that is available we always choose it.
* Otherwise, if HikariCP is available we will use it.
* If neither the Tomcat pooling datasource nor HikariCP are available and if Commons DBCP is available we will use it, but we don’t recommend it in production and its support is deprecated.
* Lastly, if Commons DBCP2 is available we will use it.

If you use the spring-boot-starter-jdbc or spring-boot-starter-data-jpa ‘starters’ you will automatically get a dependency to tomcat-jdbc.

**Note**

You can bypass that algorithm completely and specify the connection pool to use via the spring.datasource.type property. This is especially important if you are running your application in a Tomcat container as tomcat-jdbc is provided by default.

**Tip**

Additional connection pools can always be configured manually. If you define your own

DataSource bean, auto-configuration will not occur.

DataSource configuration is controlled by external configuration properties in spring.datasource.\*. For example, you might declare the following section in application.properties:

**spring.datasource.url**=jdbc:mysql://localhost/test **spring.datasource.username**=dbuser **spring.datasource.password**=dbpass **spring.datasource.driver-class-name**=com.mysql.jdbc.Driver

**Note**

You should at least specify the url using the spring.datasource.url property or Spring Boot will attempt to auto-configure an embedded database.

**Tip**

You often won’t need to specify the driver-class-name since Spring boot can deduce it for most databases from the url.

**Note**

For a pooling DataSource to be created we need to be able to verify that a valid Driver class is available, so we check for that before doing anything. I.e. if you set spring.datasource.driver-class-name=com.mysql.jdbc.Driver then that class has to be loadable.

See [DataSourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceProperties.java) for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine-tune implementation-specific settings using their respective prefix (spring.datasource.tomcat.\*, spring.datasource.hikari.\*, and spring.datasource.dbcp2.\*). Refer to the documentation of the connection pool implementation you are using for more details.

For instance, if you are using the [Tomcat connection pool](http://tomcat.apache.org/tomcat-8.0-doc/jdbc-pool.html#Common_Attributes) you could customize many additional settings:

*# Number of ms to wait before throwing an exception if no connection is available.*

**spring.datasource.tomcat.max-wait**=10000

*# Maximum number of active connections that can be allocated from this pool at the same time.*

**spring.datasource.tomcat.max-active**=50

*# Validate the connection before borrowing it from the pool.*

**spring.datasource.tomcat.test-on-borrow**=true

### Connection to a JNDI DataSource

If you are deploying your Spring Boot application to an Application Server you might want to configure and manage your DataSource using your Application Servers built-in features and access it using JNDI.

The spring.datasource.jndi-name property can be used as an alternative to the spring.datasource.url, spring.datasource.username and spring.datasource.password properties to access the DataSource from a specific JNDI location. For example, the following section in application.properties shows how you can access a JBoss AS defined DataSource:

**spring.datasource.jndi-name**=java:jboss/datasources/customers

## Using JdbcTemplate

Spring’s JdbcTemplate and NamedParameterJdbcTemplate classes are auto-configured and you can @Autowire them directly into your own beans:

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

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.stereotype.Component;

@Component

**public class** MyBean {

**private final** JdbcTemplate jdbcTemplate; @Autowired

**public** MyBean(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

*// ...*

}

## JPA and ‘Spring Data’

The Java Persistence API is a standard technology that allows you to ‘map’ objects to relational databases. The spring-boot-starter-data-jpa POM provides a quick way to get started. It provides the following key dependencies:

* Hibernate — One of the most popular JPA implementations.
* Spring Data JPA — Makes it easy to implement JPA-based repositories.
* Spring ORMs — Core ORM support from the Spring Framework.

**Tip**

We won’t go into too many details of JPA or Spring Data here. You can follow the [‘Accessing](http://spring.io/guides/gs/accessing-data-jpa/) [Data with JPA’](http://spring.io/guides/gs/accessing-data-jpa/) guide from [spring.io](http://spring.io/) and read the [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) and [Hibernate](http://hibernate.org/orm/documentation/) reference documentation.

**Note**

By default, Spring Boot uses Hibernate 5.0.x. However it’s also possible to use 4.3.x or 5.2.x if you wish. Please refer to the [Hibernate 4](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-hibernate4) and [Hibernate 5.2](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-hibernate52) samples to see how to do so.

### Entity Classes

Traditionally, JPA ‘Entity’ classes are specified in a persistence.xml file. With Spring Boot this file is not necessary and instead ‘Entity Scanning’ is used. By default all packages below your main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) will be searched.

Any classes annotated with @Entity, @Embeddable or @MappedSuperclass will be considered. A typical entity class would look something like this:

**package** com.example.myapp.domain;

**import** java.io.Serializable;

**import** javax.persistence.\*;

@Entity

**public class** City **implements** Serializable {

@Id @GeneratedValue **private** Long id;

@Column(nullable = false)

**private** String name;

@Column(nullable = false)

**private** String state;

*// ... additional members, often include @OneToMany mappings*

**protected** City() {

*// no-args constructor required by JPA spec*

*// this one is protected since it shouldn't be used directly*

}

**public** City(String name, String state) {

**this**.name = name;

**this**.country = country;

}

**public** String getName() {

**return this**.name;

}

**public** String getState() {

**return this**.state;

}

*// ... etc*

}

**Tip**

You can customize entity scanning locations using the @EntityScan annotation. See the

[*Section 77.4, “Separate @Entity definitions from Spring configuration”*](#_bookmark508) how-to.

### Spring Data JPA Repositories

Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a CityRepository interface might declare a findAllByState(String state) method to find all cities in a given state.

For more complex queries you can annotate your method using Spring Data’s [Query](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/Query.html) annotation.

Spring Data repositories usually extend from the [Repository](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/Repository.html) or [CrudRepository](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) interfaces. If you are using auto-configuration, repositories will be searched from the package containing your main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) down.

Here is a typical Spring Data repository:

**package** com.example.myapp.domain;

**import** org.springframework.data.domain.\*;

**import** org.springframework.data.repository.\*;

**public interface** CityRepository **extends** Repository<City, Long> { Page<City> findAll(Pageable pageable);

City findByNameAndCountryAllIgnoringCase(String name, String country);

}

**Tip**

We have barely scratched the surface of Spring Data JPA. For complete details check their [reference documentation](http://projects.spring.io/spring-data-jpa/).

### Creating and dropping JPA databases

By default, JPA databases will be automatically created **only** if you use an embedded database (H2, HSQL or Derby). You can explicitly configure JPA settings using spring.jpa.\* properties. For example, to create and drop tables you can add the following to your application.properties.

spring.jpa.hibernate.ddl-auto=create-drop

**Note**

Hibernate’s own internal property name for this (if you happen to remember it better) is hibernate.hbm2ddl.auto. You can set it, along with other Hibernate native properties, using spring.jpa.properties.\* (the prefix is stripped before adding them to the entity manager). Example:

spring.jpa.properties.hibernate.globally\_quoted\_identifiers=true

passes hibernate.globally\_quoted\_identifiers to the Hibernate entity manager.

By default the DDL execution (or validation) is deferred until the ApplicationContext has started. There is also a spring.jpa.generate-ddl flag, but it is not used if Hibernate autoconfig is active because the ddl-auto settings are more fine-grained.

### Open EntityManager in View

If you are running a web application, Spring Boot will by default register [OpenEntityManagerInViewInterceptor](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/orm/jpa/support/OpenEntityManagerInViewInterceptor.html) to apply the "Open EntityManager in View" pattern, i.e. to allow for lazy loading in web views. If you don’t want this behavior you should set spring.jpa.open- in-view to false in your application.properties.

## Using H2’s web console

The [H2 database](http://www.h2database.com/) provides a [browser-based console](http://www.h2database.com/html/quickstart.html#h2_console) that Spring Boot can auto-configure for you. The console will be auto-configured when the following conditions are met:

* You are developing a web application
* com.h2database:h2 is on the classpath
* You are using [Spring Boot’s developer tools](#_bookmark67)

**Tip**

If you are not using Spring Boot’s developer tools, but would still like to make use of H2’s console, then you can do so by configuring the spring.h2.console.enabled property with a value of true. The H2 console is only intended for use during development so care should be taken to ensure that spring.h2.console.enabled is not set to true in production.

### Changing the H2 console’s path

By default the console will be available at /h2-console. You can customize the console’s path using the spring.h2.console.path property.

### Securing the H2 console

When Spring Security is on the classpath and basic auth is enabled, the H2 console will be automatically secured using basic auth. The following properties can be used to customize the security configuration:

* security.user.role
* security.basic.authorize-mode
* security.basic.enabled

## Using jOOQ

Java Object Oriented Querying ([jOOQ](http://www.jooq.org/)) is a popular product from [Data Geekery](http://www.datageekery.com/) which generates Java code from your database, and lets you build type safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

### Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the [jOOQ user manual](http://www.jooq.org/doc/3.6/manual-single-page/#jooq-in-7-steps-step3). If you are using the jooq-codegen-maven plugin (and you also use the spring-boot-starter-parent “parent POM”) you can safely omit the plugin’s <version> tag. You can also use Spring Boot defined version variables (e.g. h2.version) to declare the plugin’s database dependency. Here’s an example:

**<plugin>**

**<groupId>**org.jooq**</groupId>**

**<artifactId>**jooq-codegen-maven**</artifactId>**

**<executions>**

...

**</executions>**

**<dependencies>**

**<dependency>**

**<groupId>**com.h2database**</groupId>**

**<artifactId>**h2**</artifactId>**

**<version>**${h2.version}**</version>**

**</dependency>**

**</dependencies>**

**<configuration>**

**<jdbc>**

**<driver>**org.h2.Driver**</driver>**

**<url>**jdbc:h2:~/yourdatabase**</url>**

**</jdbc>**

**<generator>**

...

**</generator>**

**</configuration>**

**</plugin>**

### Using DSLContext

The fluent API offered by jOOQ is initiated via the org.jooq.DSLContext interface. Spring Boot will auto-configure a DSLContext as a Spring Bean and connect it to your application DataSource. To use the DSLContext you can just @Autowire it:

@Component

**public class** JooqExample **implements** CommandLineRunner {

**private final** DSLContext create;

@Autowired

**public** JooqExample(DSLContext dslContext) {

**this**.create = dslContext;

}

}

**Tip**

The jOOQ manual tends to use a variable named create to hold the DSLContext, we’ve done the same for this example.

You can then use the DSLContext to construct your queries:

**public** List<GregorianCalendar> authorsBornAfter1980() {

**return this**.create.selectFrom(AUTHOR)

.where(AUTHOR.DATE\_OF\_BIRTH.greaterThan(**new** GregorianCalendar(1980, 0, 1)))

.fetch(AUTHOR.DATE\_OF\_BIRTH);

}

### Customizing jOOQ

You can customize the SQL dialect used by jOOQ by setting spring.jooq.sql-dialect in your

application.properties. For example, to specify Postgres you would add:

**spring.jooq.sql-dialect**=Postgres

More advanced customizations can be achieved by defining your own @Bean definitions which will be used when the jOOQ Configuration is created. You can define beans for the following jOOQ Types:

* ConnectionProvider
* TransactionProvider
* RecordMapperProvider
* RecordListenerProvider
* ExecuteListenerProvider
* VisitListenerProvider

You can also create your own org.jooq.Configuration @Bean if you want to take complete control of the jOOQ configuration.

# Working with NoSQL technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies including [MongoDB](http://projects.spring.io/spring-data-mongodb/), [Neo4J](http://projects.spring.io/spring-data-neo4j/), [Elasticsearch](https://github.com/spring-projects/spring-data-elasticsearch/), [Solr](http://projects.spring.io/spring-data-solr/), [Redis](http://projects.spring.io/spring-data-redis/), [Gemfire](http://projects.spring.io/spring-data-gemfire/), [Cassandra](http://projects.spring.io/spring-data-cassandra/), [Couchbase](http://projects.spring.io/spring-data-couchbase/) and [LDAP](http://projects.spring.io/spring-data-ldap/). Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr Cassandra, Couchbase and LDAP; you can make use of the other projects, but you will need to configure them yourself. Refer to the appropriate reference documentation at [projects.spring.io/spring-data](http://projects.spring.io/spring-data).

## Redis

[Redis](http://redis.io/) is a cache, message broker and richly-featured key-value store. Spring Boot offers basic auto- configuration for the [Jedis](https://github.com/xetorthio/jedis/) client library and abstractions on top of it provided by [Spring Data Redis](https://github.com/spring-projects/spring-data-redis). There is a spring-boot-starter-data-redis ‘Starter’ for collecting the dependencies in a convenient way.

### Connecting to Redis

You can inject an auto-configured RedisConnectionFactory, StringRedisTemplate or vanilla RedisTemplate instance as you would any other Spring Bean. By default the instance will attempt to connect to a Redis server using localhost:6379:

@Component

**public class** MyBean {

**private** StringRedisTemplate template; @Autowired

**public** MyBean(StringRedisTemplate template) {

**this**.template = template;

}

*// ...*

}

If you add a @Bean of your own of any of the auto-configured types it will replace the default (except in the case of RedisTemplate the exclusion is based on the bean name ‘redisTemplate’ not its type). If commons-pool2 is on the classpath you will get a pooled connection factory by default.

## MongoDB

[MongoDB](http://www.mongodb.com/) is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the spring-boot-starter-data-mongodb ‘Starter’.

### Connecting to a MongoDB database

You can inject an auto-configured org.springframework.data.mongodb.MongoDbFactory to access Mongo databases. By default the instance will attempt to connect to a MongoDB server using the URL mongodb://localhost/test:

**import** org.springframework.data.mongodb.MongoDbFactory;

**import** com.mongodb.DB;

@Component

**public class** MyBean {

**private final** MongoDbFactory mongo; @Autowired

**public** MyBean(MongoDbFactory mongo) {

**this**.mongo = mongo;

}

*// ...*

**public void** example() {

DB db = mongo.getDb();

*// ...*

}

}

You can set spring.data.mongodb.uri property to change the URL and configure additional settings such as the *replica set*:

**spring.data.mongodb.uri**=mongodb://user:secret@mongo1.example.com:12345,mongo2.example.com:23456/test

Alternatively, as long as you’re using Mongo 2.x, specify a host/port. For example, you might declare the following in your application.properties:

**spring.data.mongodb.host**=mongoserver **spring.data.mongodb.port**=27017

**Note**

spring.data.mongodb.host and spring.data.mongodb.port are not supported if you’re using the Mongo 3.0 Java driver. In such cases, spring.data.mongodb.uri should be used to provide all of the configuration.

**Tip**

If spring.data.mongodb.port is not specified the default of 27017 is used. You could simply delete this line from the sample above.

**Tip**

If you aren’t using Spring Data Mongo you can inject com.mongodb.Mongo beans instead of using MongoDbFactory.

You can also declare your own MongoDbFactory or Mongo bean if you want to take complete control of establishing the MongoDB connection.

### MongoTemplate

Spring Data Mongo provides a [MongoTemplate](http://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/MongoTemplate.html) class that is very similar in its design to Spring’s

JdbcTemplate. As with JdbcTemplate Spring Boot auto-configures a bean for you to simply inject:

**import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.data.mongodb.core.MongoTemplate; **import** org.springframework.stereotype.Component;

@Component

**public class** MyBean {

**private final** MongoTemplate mongoTemplate;

@Autowired

**public** MyBean(MongoTemplate mongoTemplate) {

**this**.mongoTemplate = mongoTemplate;

}

*// ...*

}

See the MongoOperations Javadoc for complete details.

### Spring Data MongoDB repositories

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure; so you could take the JPA example from earlier and, assuming that City is now a Mongo data class rather than a JPA @Entity, it will work in the same way.

**package** com.example.myapp.domain;

**import** org.springframework.data.domain.\*;

**import** org.springframework.data.repository.\*;

**public interface** CityRepository **extends** Repository<City, Long> { Page<City> findAll(Pageable pageable);

City findByNameAndCountryAllIgnoringCase(String name, String country);

}

**Tip**

For complete details of Spring Data MongoDB, including its rich object mapping technologies, refer to their [reference documentation](http://projects.spring.io/spring-data-mongodb/).

### Embedded Mongo

Spring Boot offers auto-configuration for [Embedded Mongo](https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo). To use it in your Spring Boot application add a dependency on de.flapdoodle.embed:de.flapdoodle.embed.mongo.

The port that Mongo will listen on can be configured using the spring.data.mongodb.port property. To use a randomly allocated free port use a value of zero. The MongoClient created by MongoAutoConfiguration will be automatically configured to use the randomly allocated port.

**Note**

If you do not configure a custom port, the embedded support will use a random port by default (rather than 27017).

If you have SLF4J on the classpath, output produced by Mongo will be automatically routed to a logger named org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo.

You can declare your own IMongodConfig and IRuntimeConfig beans to take control of the Mongo instance’s configuration and logging routing.

## Neo4j

[Neo4j](http://neo4j.com/) is an open-source NoSQL graph database that uses a rich data model of nodes related by first class relationships which is better suited for connected big data than traditional rdbms approaches. Spring Boot offers several conveniences for working with Neo4j, including the spring- boot-starter-data-neo4j ‘Starter’.

### Connecting to a Neo4j database

You can inject an auto-configured Neo4jSession, Session or Neo4jOperations instance as you would any other Spring Bean. By default the instance will attempt to connect to a Neo4j server using localhost:7474:

@Component

**public class** MyBean {

**private final** Neo4jTemplate neo4jTemplate; @Autowired

**public** MyBean(Neo4jTemplate neo4jTemplate) {

**this**.neo4jTemplate = neo4jTemplate;

}

*// ...*

}

You can take full control of the configuration by adding a org.neo4j.ogm.config.Configuration @Bean of your own. Also, adding a @Bean of type Neo4jOperations disables the auto-configuration.

You can configure the user and credentials to use via the spring.data.neo4j.\* properties:

**spring.data.neo4j.uri**=http://my-server:7474 **spring.data.neo4j.username**=neo4j **spring.data.neo4j.password**=secret

### Using the embedded mode

If you add org.neo4j:neo4j-ogm-embedded-driver to the dependencies of your application, Spring Boot will automatically configure an in-process embedded instance of Neo4j that will not persist any data when your application shuts down. You can explicitly disable that mode using spring.data.neo4j.embedded.enabled=false. You can also enable persistence for the embedded mode:

spring.data.neo4j.uri=file://var/tmp/graph.db

**Note**

The Neo4j OGM embedded driver does not provide the Neo4j kernel. Users are expected to provide this dependency manually, see [the documentation](http://neo4j.com/docs/ogm-manual/current/reference/#reference%3Agetting-started) for more details.

### Neo4jSession

By default, if you are running a web application, the session is bound to the thread for the entire processing of the request (i.e. the "Open Session in View" pattern). If you don’t want this behavior add the following to your application.properties:

spring.data.neo4j.open-in-view=false

### Spring Data Neo4j repositories

Spring Data includes repository support for Neo4j.

In fact, both Spring Data JPA and Spring Data Neo4j share the same common infrastructure; so you could take the JPA example from earlier and, assuming that City is now a Neo4j OGM @NodeEntity rather than a JPA @Entity, it will work in the same way.

**Tip**

You can customize entity scanning locations using the @EntityScan annotation.

To enable repository support (and optionally support for @Transactional), add the following two annotations to your Spring configuration:

@EnableNeo4jRepositories(basePackages = "com.example.myapp.repository") @EnableTransactionManagement

### Repository example

**package** com.example.myapp.domain;

**import** org.springframework.data.domain.\*;

**import** org.springframework.data.repository.\*;

**public interface** CityRepository **extends** GraphRepository<City> { Page<City> findAll(Pageable pageable);

City findByNameAndCountry(String name, String country);

}

**Tip**

For complete details of Spring Data Neo4j, including its rich object mapping technologies, refer to their [reference documentation](http://projects.spring.io/spring-data-neo4j/).

## Gemfire

[Spring Data Gemfire](https://github.com/spring-projects/spring-data-gemfire) provides convenient Spring-friendly tools for accessing the [Pivotal Gemfire](http://pivotal.io/big-data/pivotal-gemfire#details) data management platform. There is a spring-boot-starter-data-gemfire ‘Starter’ for collecting the dependencies in a convenient way. There is currently no auto-configuration support for Gemfire, but you can enable Spring Data Repositories with a [single annotation (@EnableGemfireRepositories)](https://github.com/spring-projects/spring-data-gemfire/blob/master/src/main/java/org/springframework/data/gemfire/repository/config/EnableGemfireRepositories.java).

## Solr

[Apache Solr](http://lucene.apache.org/solr/) is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 client library and abstractions on top of it provided by [Spring Data Solr](https://github.com/spring-projects/spring-data-solr). There is a spring-boot-starter-data- solr ‘Starter’ for collecting the dependencies in a convenient way.

### Connecting to Solr

You can inject an auto-configured SolrClient instance as you would any other Spring bean. By default the instance will attempt to connect to a server using localhost:8983/solr:

@Component

**public class** MyBean {

**private** SolrClient solr; @Autowired

**public** MyBean(SolrClient solr) {

**this**.solr = solr;

}

*// ...*

}

If you add a @Bean of your own of type SolrClient it will replace the default.

### Spring Data Solr repositories

Spring Data includes repository support for Apache Solr. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Solr share the same common infrastructure; so you could take the JPA example from earlier and, assuming that City is now a @SolrDocument class rather than a JPA @Entity, it will work in the same way.

**Tip**

For complete details of Spring Data Solr, refer to their [reference documentation](http://projects.spring.io/spring-data-solr/).

## Elasticsearch

[Elasticsearch](http://www.elasticsearch.org/) is an open source, distributed, real-time search and analytics engine. Spring Boot offers basic auto-configuration for the Elasticsearch and abstractions on top of it provided by [Spring Data](https://github.com/spring-projects/spring-data-elasticsearch) [Elasticsearch](https://github.com/spring-projects/spring-data-elasticsearch). There is a spring-boot-starter-data-elasticsearch ‘Starter’ for collecting the dependencies in a convenient way. Spring Boot also supports [Jest](https://github.com/searchbox-io/Jest).

### Connecting to Elasticsearch using Jest

If you have Jest on the classpath, you can inject an auto-configured JestClient targeting

localhost:9200 by default. You can further tune how the client is configured:

**spring.elasticsearch.jest.uris**=http://search.example.com:9200 **spring.elasticsearch.jest.read-timeout**=10000 **spring.elasticsearch.jest.username**=user **spring.elasticsearch.jest.password**=secret

You can also register an arbitrary number of beans implementing HttpClientConfigBuilderCustomizer for more advanced customizations. The example below tunes additional HTTP settings:

**static class** HttpSettingsCustomizer **implements** HttpClientConfigBuilderCustomizer {

@Override

**public void** customize(HttpClientConfig.Builder builder) { builder.maxTotalConnection(100).defaultMaxTotalConnectionPerRoute(5);

}

}

To take full control over the registration, define a JestClient bean.

### Connecting to Elasticsearch using Spring Data

You can inject an auto-configured ElasticsearchTemplate or Elasticsearch Client instance as you would any other Spring Bean. By default the instance will embed a local in-memory server (a Node in Elasticsearch terms) and use the current working directory as the home directory for the server. In this setup, the first thing to do is to tell Elasticsearch where to store its files:

**spring.data.elasticsearch.properties.path.home**=/foo/bar

Alternatively, you can switch to a remote server (i.e. a TransportClient) by setting

spring.data.elasticsearch.cluster-nodes to a comma-separated ‘host:port’ list.

**spring.data.elasticsearch.cluster-nodes**=localhost:9300

@Component

**public class** MyBean {

**private** ElasticsearchTemplate template; @Autowired

**public** MyBean(ElasticsearchTemplate template) {

**this**.template = template;

}

*// ...*

}

If you add a @Bean of your own of type ElasticsearchTemplate it will replace the default.

### Spring Data Elasticsearch repositories

Spring Data includes repository support for Elasticsearch. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure; so you could take the JPA example from earlier and, assuming that City is now an Elasticsearch @Document class rather than a JPA @Entity, it will work in the same way.

**Tip**

For complete details of Spring Data Elasticsearch, refer to their [reference documentation](http://docs.spring.io/spring-data/elasticsearch/docs/).

## Cassandra

[Cassandra](http://cassandra.apache.org/) is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and abstractions on top of it provided by [Spring Data Cassandra](https://github.com/spring-projects/spring-data-cassandra). There is a spring-boot-starter- data-cassandra ‘Starter’ for collecting the dependencies in a convenient way.

### Connecting to Cassandra

You can inject an auto-configured CassandraTemplate or a Cassandra Session instance as you would with any other Spring Bean. The spring.data.cassandra.\* properties can be used to customize the connection. Generally you will provide keyspace-name and contact-points properties:

**spring.data.cassandra.keyspace-name**=mykeyspace **spring.data.cassandra.contact-points**=cassandrahost1,cassandrahost2

@Component

**public class** MyBean {

**private** CassandraTemplate template; @Autowired

**public** MyBean(CassandraTemplate template) {

**this**.template = template;

}

*// ...*

}

If you add a @Bean of your own of type CassandraTemplate it will replace the default.

### Spring Data Cassandra repositories

Spring Data includes basic repository support for Cassandra. Currently this is more limited than the JPA repositories discussed earlier, and will need to annotate finder methods with @Query.

**Tip**

For complete details of Spring Data Cassandra, refer to their [reference documentation](http://docs.spring.io/spring-data/cassandra/docs/).

## Couchbase

[Couchbase](http://www.couchbase.com/) is an open-source, distributed multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and abstractions on top of it provided by [Spring Data Couchbase](https://github.com/spring-projects/spring-data-couchbase). There is a spring-boot-starter- data-couchbase ‘Starter’ for collecting the dependencies in a convenient way.

### Connecting to Couchbase

You can very easily get a Bucket and Cluster by adding the Couchbase SDK and some configuration. The spring.couchbase.\* properties can be used to customize the connection. Generally you will provide the bootstrap hosts, bucket name and password:

**spring.couchbase.bootstrap-hosts**=my-host-1,192.168.1.123 **spring.couchbase.bucket.name**=my-bucket **spring.couchbase.bucket.password**=secret

**Tip**

You need to provide *at least* the bootstrap host(s), in which case the bucket name is default and the password is the empty String. Alternatively, you can define your

own org.springframework.data.couchbase.config.CouchbaseConfigurer @Bean

to take control over the whole configuration.

It is also possible to customize some of the CouchbaseEnvironment settings. For instance the following configuration changes the timeout to use to open a new Bucket and enables SSL support:

**spring.couchbase.env.timeouts.connect**=3000 **spring.couchbase.env.ssl.key-store**=/location/of/keystore.jks **spring.couchbase.env.ssl.key-store-password**=secret

Check the spring.couchbase.env.\* properties for more details.

### Spring Data Couchbase repositories

Spring Data includes repository support for Couchbase. For complete details of Spring Data Couchbase, refer to their [reference documentation](http://docs.spring.io/spring-data/couchbase/docs/current/reference/html/).

You can inject an auto-configured CouchbaseTemplate instance as you would with any other Spring Bean as long as a *default* CouchbaseConfigurer is available (that happens when you enable the couchbase support as explained above).

@Component

**public class** MyBean {

**private final** CouchbaseTemplate template; @Autowired

**public** MyBean(CouchbaseTemplate template) {

**this**.template = template;

}

*// ...*

}

There are a few beans that you can define in your own configuration to override those provided by the auto-configuration:

* A CouchbaseTemplate @Bean with name couchbaseTemplate
* An IndexManager @Bean with name couchbaseIndexManager
* A CustomConversions @Bean with name couchbaseCustomConversions

To avoid hard-coding those names in your own config, you can reuse BeanNames provided by Spring Data Couchbase. For instance, you can customize the converters to use as follows:

@Configuration

**public class** SomeConfiguration {

@Bean(BeanNames.COUCHBASE\_CUSTOM\_CONVERSIONS)

**public** CustomConversions myCustomConversions() {

**return new** CustomConversions(...);

}

*// ...*

}

**Tip**

If you want to fully bypass the auto-configuration for Spring Data Couchbase, provide your own org.springframework.data.couchbase.config.AbstractCouchbaseDataConfiguration implementation.

## LDAP

[LDAP](https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol) (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. Spring Boot offers auto-configuration for any compliant LDAP server as well as support for the embedded in-memory LDAP server from [UnboundID](https://www.ldap.com/unboundid-ldap-sdk-for-java).

LDAP abstractions are provided by [Spring Data LDAP](https://github.com/spring-projects/spring-data-ldap). There is a spring-boot-starter-data- ldap ‘Starter’ for collecting the dependencies in a convenient way.

### Connecting to an LDAP server

To connect to an LDAP server make sure you declare a dependency on the spring-boot- starter-data-ldap ‘Starter’ or spring-ldap-core then declare the URLs of your server in your application.properties:

**spring.ldap.urls**=ldap://myserver:1235 **spring.ldap.username**=admin **spring.ldap.password**=secret

If you need to customize connection settings you can use the spring.ldap.base and

spring.ldap.base-environment properties.

### Spring Data LDAP repositories

Spring Data includes repository support for LDAP. For complete details of Spring Data LDAP, refer to their [reference documentation](http://docs.spring.io/spring-data/ldap/docs/1.0.x/reference/html/).

You can also inject an auto-configured LdapTemplate instance as you would with any other Spring Bean.

@Component

**public class** MyBean {

**private final** LdapTemplate template; @Autowired

**public** MyBean(LdapTemplate template) {

**this**.template = template;

}

*// ...*

}

### Embedded in-memory LDAP server

For testing purposes Spring Boot supports auto-configuration of an in-memory LDAP server from [UnboundID](https://www.ldap.com/unboundid-ldap-sdk-for-java). To configure the server add a dependency to com.unboundid:unboundid-ldapsdk and declare a base-dn property:

**spring.ldap.embedded.base-dn**=dc=spring,dc=io

By default the server will start on a random port and they trigger the regular LDAP support (there is no need to specify a spring.ldap.urls property).

If there is a schema.ldif file on your classpath it will be used to initialize the server. You can also use the spring.ldap.embedded.ldif property if you want to load the initialization script from a different resource.

By default, a standard schema will be used to validate LDIF files, you can turn off validation altogether using the spring.ldap.embedded.validation.enabled property. If you have custom attributes, you can use spring.ldap.embedded.validation.schema to define your custom attribute types or object classes.

# Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, reducing thus the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache infrastructure as long as the caching support is enabled via the @EnableCaching annotation.

**Note**

Check the [relevant section](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#cache) of the Spring Framework reference for more details.

In a nutshell, adding caching to an operation of your service is as easy as adding the relevant annotation to its method:

**import** org.springframework.cache.annotation.Cacheable

**import** org.springframework.stereotype.Component;

@Component

**public class** MathService {

@Cacheable("piDecimals")

**public int** computePiDecimal(**int** i) {

*// ...*

}

}

This example demonstrates the use of caching on a potentially costly operation. Before invoking computePiDecimal, the abstraction will look for an entry in the piDecimals cache matching the i argument. If an entry is found, the content in the cache is immediately returned to the caller and the method is not invoked. Otherwise, the method is invoked and the cache is updated before returning the value.

**Note**

You can also use the standard JSR-107 (JCache) annotations (e.g. @CacheResult) transparently. We strongly advise you however to not mix and match them.

If you do not add any specific cache library, Spring Boot will auto-configure a [Simple provider](#_bookmark226) that uses concurrent maps in memory. When a cache is required (i.e. piDecimals in the example above), this provider will create it on-the-fly for you. The simple provider is not really recommended for production usage, but it’s great for getting started and making sure that you understand the features. When you have made up your mind about the cache provider to use, please make sure to read its documentation to figure out how to configure the caches that your application uses. Practically all providers require you to explicitly configure every cache that you use in the application. Some offer a way to customize the default caches defined by the spring.cache.cache-names property.

**Tip**

It is also possible to [update](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#cache-annotations-put) or [evict](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#cache-annotations-evict) data from the cache transparently.

**Note**

If you are using the cache infrastructure with beans that are not interface-based, make sure to enable the proxyTargetClass attribute of @EnableCaching.

## Supported cache providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the org.springframework.cache.Cache and org.springframework.cache.CacheManager interfaces.

If you haven’t defined a bean of type CacheManager or a CacheResolver named cacheResolver

(see CachingConfigurer), Spring Boot tries to detect the following providers (in this order):

* [Generic](#_bookmark217)
* [JCache (JSR-107)](#_bookmark218) (EhCache 3, Hazelcast, Infinispan, etc)
* [EhCache 2.x](#_bookmark219)
* [Hazelcast](#_bookmark220)
* [Infinispan](#_bookmark221)
* [Couchbase](#_bookmark222)
* [Redis](#_bookmark223)
* [Caffeine](#_bookmark224)
* [Guava](#_bookmark225) (deprecated)
* [Simple](#_bookmark226)

**Tip**

It is also possible to *force* the cache provider to use via the spring.cache.type property. Use this property if you need to [disable caching altogether](#_bookmark227) in certain environment (e.g. tests).

**Tip**

Use the spring-boot-starter-cache ‘Starter’ to quickly add basic caching dependencies. The starter brings in spring-context-support: if you are adding dependencies manually, you must include spring-context-support in order to use the JCache, EhCache 2.x or Guava support.

If the CacheManager is auto-configured by Spring Boot, you can further tune its configuration before it is fully initialized by exposing a bean implementing the CacheManagerCustomizer interface. The following sets a flag to say that null values should be passed down to the underlying map.

@Bean

**public** CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {

**return new** CacheManagerCustomizer<ConcurrentMapCacheManager>() { @Override

**public void** customize(ConcurrentMapCacheManager cacheManager) { cacheManager.setAllowNullValues(false);

}

};

}

**Note**

In the example above, an auto-configured ConcurrentMapCacheManager is expected. If that is not the case (either you provided your own config or a different cache provider was auto- configured), the customizer won’t be invoked at all. You can have as many customizers as you want and you can also order them as usual using @Order or Ordered.

### Generic

Generic caching is used if the context defines *at least* one org.springframework.cache.Cache

bean. A CacheManager wrapping all beans of that type is created.

### JCache (JSR-107)

JCache is bootstrapped via the presence of a javax.cache.spi.CachingProvider on the classpath (i.e. a JSR-107 compliant caching library) and the JCacheCacheManager provided by the spring-boot-starter-cache ‘Starter’. There are various compliant libraries out there and Spring Boot provides dependency management for Ehcache 3, Hazelcast and Infinispan. Any other compliant library can be added as well.

It might happen that more than one provider is present, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not enforce a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate with implementation details.

*# Only necessary if more than one provider is present* **spring.cache.jcache.provider**=com.acme.MyCachingProvider **spring.cache.jcache.config**=classpath:acme.xml

**Note**

Since a cache library may offer both a native implementation and JSR-107 support Spring Boot will prefer the JSR-107 support so that the same features are available if you switch to a different JSR-107 implementation.

**Tip**

Spring Boot has a [general support for Hazelcast](#_bookmark254). If a single HazelcastInstance is available, it is automatically reused for the CacheManager as well unless the spring.cache.jcache.config property is specified.

There are several ways to customize the underlying javax.cache.cacheManager:

* Caches can be created on startup via the spring.cache.cache-names property. If a custom

javax.cache.configuration.Configuration bean is defined, it is used to customize them.

* org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer beans are invoked with the reference of the CacheManager for full customization.

**Tip**

If a standard javax.cache.CacheManager bean is defined, it is wrapped automatically in a org.springframework.cache.CacheManager implementation that the abstraction expects. No further customization is applied on it.

### EhCache 2.x

EhCache 2.x is used if a file named ehcache.xml can be found at the root of the classpath. If EhCache 2.x, the EhCacheCacheManager provided by the spring-boot-starter-cache ‘Starter’ and such file is present it is used to bootstrap the cache manager. An alternate configuration file can be provided as well using:

**spring.cache.ehcache.config**=classpath:config/another-config.xml

### Hazelcast

Spring Boot has a [general support for Hazelcast](#_bookmark254). If a HazelcastInstance has been auto-configured, it is automatically wrapped in a CacheManager.

### Infinispan

Infinispan has no default configuration file location so it must be specified explicitly (or the default bootstrap is used).

**spring.cache.infinispan.config**=infinispan.xml

Caches can be created on startup via the spring.cache.cache-names property. If a custom

ConfigurationBuilder bean is defined, it is used to customize them.

**Note**

The support of Infinispan in Spring Boot is restricted to the embedded mode and is quite basic. If you want more options you should use the official Infinispan Spring Boot starter instead, check [the documentation](https://github.com/infinispan/infinispan-spring-boot) for more details.

### Couchbase

If the Couchbase java client and the couchbase-spring-cache implementation are available and Couchbase is [configured](#_bookmark208), a CouchbaseCacheManager will be auto-configured. It is also possible to create additional caches on startup using the spring.cache.cache-names property. These will operate on the Bucket that was auto-configured. You can *also* create additional caches on another Bucket using the customizer: assume you need two caches on the "main" Bucket (foo and bar) and one biz cache with a custom time to live of 2sec on the another Bucket. First, you can create the two first caches simply via configuration:

**spring.cache.cache-names**=foo,bar

Then define this extra @Configuration to configure the extra Bucket and the biz cache:

@Configuration

**public class** CouchbaseCacheConfiguration {

**private final** Cluster cluster;

**public** CouchbaseCacheConfiguration(Cluster cluster) {

**this**.cluster = cluster;

}

@Bean

**public** Bucket anotherBucket() {

**return this**.cluster.openBucket(***"another"***, ***"secret"***);

}

@Bean

**public** CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer() {

**return** c -> {

c.prepareCache(***"biz"***, CacheBuilder.newInstance(anotherBucket())

.withExpiration(2));

};

}

}

This sample configuration reuses the Cluster that was created via auto-configuration.

### Redis

If Redis is available and configured, the RedisCacheManager is auto-configured. It is also possible to create additional caches on startup using the spring.cache.cache-names property.

**Note**

By default, a key prefix is added to prevent that if two separate caches use the same key, Redis would have overlapping keys and be likely to return invalid values. We strongly recommend to keep this setting enabled if you create your own RedisCacheManager.

### Caffeine

Caffeine is a Java 8 rewrite of Guava’s cache and will supersede the Guava support in Spring Boot 2.0. If Caffeine is present, a CaffeineCacheManager (provided by the spring-boot-starter-cache ‘Starter’) is auto-configured. Caches can be created on startup using the spring.cache.cache- names property and customized by one of the following (in this order):

1. A cache spec defined by spring.cache.caffeine.spec
2. A com.github.benmanes.caffeine.cache.CaffeineSpec bean is defined
3. A com.github.benmanes.caffeine.cache.Caffeine bean is defined

For instance, the following configuration creates a foo and bar caches with a maximum size of 500 and a *time to live* of 10 minutes

**spring.cache.cache-names**=foo,bar **spring.cache.caffeine.spec**=maximumSize=500,expireAfterAccess=600s

Besides, if a com.github.benmanes.caffeine.cache.CacheLoader bean is defined, it is automatically associated to the CaffeineCacheManager. Since the CacheLoader is going to be associated to *all* caches managed by the cache manager, it must be defined as CacheLoader<Object, Object>. Any other generic type will be ignored by the auto-configuration.

### Guava (deprecated)

If Guava is present, a GuavaCacheManager is auto-configured. Caches can be created on startup using the spring.cache.cache-names property and customized by one of the following (in this order):

1. A cache spec defined by spring.cache.guava.spec
2. A com.google.common.cache.CacheBuilderSpec bean is defined
3. A com.google.common.cache.CacheBuilder bean is defined

For instance, the following configuration creates a foo and bar caches with a maximum size of 500 and a *time to live* of 10 minutes

**spring.cache.cache-names**=foo,bar **spring.cache.guava.spec**=maximumSize=500,expireAfterAccess=600s

Besides, if a com.google.common.cache.CacheLoader bean is defined, it is automatically associated to the GuavaCacheManager. Since the CacheLoader is going to be associated to *all* caches managed by the cache manager, it must be defined as CacheLoader<Object, Object>. Any other generic type will be ignored by the auto-configuration.

### Simple

If none of the other providers can be found, a simple implementation using a ConcurrentHashMap as cache store is configured. This is the default if no caching library is present in your application. Caches are created on-the-fly by default but you can restrict the list of available caches using the cache-names property. For instance, if you want only foo and bar caches:

**spring.cache.cache-names**=foo,bar

If you do this and your application uses a cache not listed then it will fail at runtime when the cache is needed, but not on startup. This is similar to the way the "real" cache providers behave if you use an undeclared cache.

### None

When @EnableCaching is present in your configuration, a suitable cache configuration is expected as well. If you need to disable caching altogether in certain environments, force the cache type to none to use a no-op implementation:

**spring.cache.type**=none

# Messaging

The Spring Framework provides extensive support for integrating with messaging systems: from simplified use of the JMS API using JmsTemplate to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the ‘Advanced Message Queuing Protocol’ and Spring Boot also provides auto-configuration options for RabbitTemplate and RabbitMQ. There is also support for STOMP messaging natively in Spring WebSocket and Spring Boot has support for that through starters and a small amount of auto-configuration. Spring Boot also has support for Apache Kafka.

## JMS

The javax.jms.ConnectionFactory interface provides a standard method of creating a javax.jms.Connection for interacting with a JMS broker. Although Spring needs a ConnectionFactory to work with JMS, you generally won’t need to use it directly yourself and you can instead rely on higher level messaging abstractions (see the [relevant section](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#jms) of the Spring Framework reference documentation for details). Spring Boot also auto-configures the necessary infrastructure to send and receive messages.

### ActiveMQ support

Spring Boot can also configure a ConnectionFactory when it detects that ActiveMQ is available on the classpath. If the broker is present, an embedded broker is started and configured automatically (as long as no broker URL is specified through configuration).

**Note**

If you are using spring-boot-starter-activemq the necessary dependencies to connect or embed an ActiveMQ instance are provided, as well as the Spring infrastructure to integrate with JMS.

ActiveMQ configuration is controlled by external configuration properties in spring.activemq.\*. For example, you might declare the following section in application.properties:

**spring.activemq.broker-url**=tcp://192.168.1.210:9876 **spring.activemq.user**=admin **spring.activemq.password**=secret

You can also pool JMS resources by adding a dependency to org.apache.activemq:activemq- pool and configure the PooledConnectionFactory accordingly:

**spring.activemq.pool.enabled**=true **spring.activemq.pool.max-connections**=50

**Tip**

See [ActiveMQProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/activemq/ActiveMQProperties.java) for more of the supported options. You can also register an arbitrary number of beans implementing ActiveMQConnectionFactoryCustomizer for more advanced customizations.

By default, ActiveMQ creates a destination if it does not exist yet, so destinations are resolved against their provided names.

### Artemis support

Spring Boot can auto-configure a ConnectionFactory when it detects that Artemis is available on the classpath. If the broker is present, an embedded broker is started and configured automatically (unless the mode property has been explicitly set). The supported modes are: embedded (to make explicit that an embedded broker is required and should lead to an error if the broker is not available in the classpath), and native to connect to a broker using the netty transport protocol. When the latter is configured, Spring Boot configures a ConnectionFactory connecting to a broker running on the local machine with the default settings.

**Note**

If you are using spring-boot-starter-artemis the necessary dependencies to connect to an existing Artemis instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding org.apache.activemq:artemis-jms-server to your application allows you to use the embedded mode.

Artemis configuration is controlled by external configuration properties in spring.artemis.\*. For example, you might declare the following section in application.properties:

**spring.artemis.mode**=native **spring.artemis.host**=192.168.1.210 **spring.artemis.port**=9876 **spring.artemis.user**=admin **spring.artemis.password**=secret

When embedding the broker, you can choose if you want to enable persistence, and the list of destinations that should be made available. These can be specified as a comma-separated list to create them with the default options; or you can define bean(s) of type org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration or org.apache.activemq.artemis.jms.server.config.TopicConfiguration, for advanced queue and topic configurations respectively.

See [ArtemisProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/artemis/ArtemisProperties.java) for more of the supported options.

No JNDI lookup is involved at all and destinations are resolved against their names, either using the ‘name’ attribute in the Artemis configuration or the names provided through configuration.

### Using a JNDI ConnectionFactory

If you are running your application in an Application Server Spring Boot will attempt to locate a JMS ConnectionFactory using JNDI. By default the locations java:/JmsXA and java:/ XAConnectionFactory will be checked. You can use the spring.jms.jndi-name property if you need to specify an alternative location:

**spring.jms.jndi-name**=java:/MyConnectionFactory

### Sending a message

Spring’s JmsTemplate is auto-configured and you can autowire it directly into your own beans:

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

**import** org.springframework.jms.core.JmsTemplate;

**import** org.springframework.stereotype.Component;

@Component

**public class** MyBean {

**private final** JmsTemplate jmsTemplate; @Autowired

**public** MyBean(JmsTemplate jmsTemplate) {

**this**.jmsTemplate = jmsTemplate;

}

*// ...*

}

**Note**

[JmsMessagingTemplate](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/jms/core/JmsMessagingTemplate.html) can be injected in a similar manner. If a DestinationResolver or MessageConverter beans are defined, they are associated automatically to the auto-configured JmsTemplate.

### Receiving a message

When the JMS infrastructure is present, any bean can be annotated with @JmsListener to create a listener endpoint. If no JmsListenerContainerFactory has been defined, a default one is configured automatically. If a DestinationResolver or MessageConverter beans are defined, they are associated automatically to the default factory.

The default factory is transactional by default. If you are running in an infrastructure where a JtaTransactionManager is present, it will be associated to the listener container by default. If not, the sessionTransacted flag will be enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding @Transactional on your listener method (or a delegate thereof). This will make sure that the incoming message is acknowledged once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

The following component creates a listener endpoint on the someQueue destination:

@Component

**public class** MyBean {

@JmsListener(destination = "someQueue")

**public void** processMessage(String content) {

*// ...*

}

}

**Tip**

Check [the Javadoc of @EnableJms](http://docs.spring.io/spring/docs/4.3.12.RELEASE/javadoc-api/org/springframework/jms/annotation/EnableJms.html) for more details.

If you need to create more JmsListenerContainerFactory instances or if you want to override the default, Spring Boot provides a DefaultJmsListenerContainerFactoryConfigurer that you can use to initialize a DefaultJmsListenerContainerFactory with the same settings as the one that is auto-configured.

For instance, the following exposes another factory that uses a specific MessageConverter:

@Configuration

**static class** JmsConfiguration {

@Bean

**public** DefaultJmsListenerContainerFactory myFactory( DefaultJmsListenerContainerFactoryConfigurer configurer) {

DefaultJmsListenerContainerFactory factory =

**new** DefaultJmsListenerContainerFactory(); configurer.configure(factory, connectionFactory()); factory.setMessageConverter(myMessageConverter()); **return** factory;

}

}

Then you can use in any @JmsListener-annotated method as follows:

@Component

**public class** MyBean {

@JmsListener(destination = ***"someQueue"***, **containerFactory="myFactory"**) **public void** processMessage(String content) {

*// ...*

}

}

## AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. Spring Boot offers several conveniences for working with AMQP via RabbitMQ, including the spring-boot-starter-amqp ‘Starter’.

### RabbitMQ support

RabbitMQ is a lightweight, reliable, scalable and portable message broker based on the AMQP protocol. Spring uses RabbitMQ to communicate using the AMQP protocol.

RabbitMQ configuration is controlled by external configuration properties in spring.rabbitmq.\*. For example, you might declare the following section in application.properties:

**spring.rabbitmq.host**=localhost **spring.rabbitmq.port**=5672 **spring.rabbitmq.username**=admin **spring.rabbitmq.password**=secret

See [RabbitProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/amqp/RabbitProperties.java) for more of the supported options.

**Tip**

Check [Understanding AMQP, the protocol used by RabbitMQ](http://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/) for more details.

### Sending a message

Spring’s AmqpTemplate and AmqpAdmin are auto-configured and you can autowire them directly into your own beans:

**import** org.springframework.amqp.core.AmqpAdmin;

**import** org.springframework.amqp.core.AmqpTemplate;

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

**import** org.springframework.stereotype.Component;

@Component

**public class** MyBean {

**private final** AmqpAdmin amqpAdmin;

**private final** AmqpTemplate amqpTemplate;

@Autowired

**public** MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {

**this**.amqpAdmin = amqpAdmin;

**this**.amqpTemplate = amqpTemplate;

}

*// ...*

}

**Note**

[RabbitMessagingTemplate](http://docs.spring.io/spring-amqp/docs/current/api/org/springframework/amqp/rabbit/core/RabbitMessagingTemplate.html) can be injected in a similar manner. If a MessageConverter

bean is defined, it is associated automatically to the auto-configured AmqpTemplate.

Any org.springframework.amqp.core.Queue that is defined as a bean will be automatically used to declare a corresponding queue on the RabbitMQ instance if necessary.

You can enable retries on the AmqpTemplate to retry operations, for example in the event the broker connection is lost. Retries are disabled by default.

### Receiving a message

When the Rabbit infrastructure is present, any bean can be annotated with @RabbitListener to create a listener endpoint. If no RabbitListenerContainerFactory has been defined, a default one is configured automatically. If a MessageConverter or MessageRecoverer beans are defined, they are associated automatically to the default factory.

The following component creates a listener endpoint on the someQueue queue:

@Component

**public class** MyBean {

@RabbitListener(queues = "someQueue")

**public void** processMessage(String content) {

*// ...*

}

}

**Tip**

Check [the Javadoc of @EnableRabbit](http://docs.spring.io/spring-amqp/docs/current/api/org/springframework/amqp/rabbit/annotation/EnableRabbit.html) for more details.

If you need to create more RabbitListenerContainerFactory instances or if you want to override the default, Spring Boot provides a SimpleRabbitListenerContainerFactoryConfigurer that you can use to initialize a SimpleRabbitListenerContainerFactory with the same settings as the one that is auto-configured.

For instance, the following exposes another factory that uses a specific MessageConverter:

@Configuration

**static class** RabbitConfiguration {

@Bean

**public** SimpleRabbitListenerContainerFactory myFactory( SimpleRabbitListenerContainerFactoryConfigurer configurer) {

SimpleRabbitListenerContainerFactory factory =

**new** SimpleRabbitListenerContainerFactory(); configurer.configure(factory, connectionFactory); factory.setMessageConverter(myMessageConverter()); **return** factory;

}

}

Then you can use in any @RabbitListener-annotated method as follows:

@Component

**public class** MyBean {

@RabbitListener(queues = ***"someQueue"***, **containerFactory="myFactory"**) **public void** processMessage(String content) {

*// ...*

}

}

You can enable retries to handle situations where your listener throws an exception. By default RejectAndDontRequeueRecoverer is used but you can define a MessageRecoverer of your own. When retries are exhausted, the message will be rejected and either dropped or routed to a dead-letter exchange if the broker is configured so. Retries are disabled by default.

**Important**

If retries are not enabled and the listener throws an exception, by default the delivery will be retried indefinitely. You can modify this behavior in two ways; set the defaultRequeueRejected property to false and zero re-deliveries will be attempted; or, throw an AmqpRejectAndDontRequeueException to signal the message should be rejected. This is the mechanism used when retries are enabled and the maximum delivery attempts are reached.

## Apache Kafka Support

[Apache Kafka](http://kafka.apache.org/) is supported by providing auto-configuration of the spring-kafka project.

Kafka configuration is controlled by external configuration properties in spring.kafka.\*. For example, you might declare the following section in application.properties:

**spring.kafka.bootstrap-servers**=localhost:9092 **spring.kafka.consumer.group-id**=myGroup

See [KafkaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/kafka/KafkaProperties.java) for more of the supported options.

### Sending a Message

Spring’s KafkaTemplate is auto-configured and you can autowire them directly in your own beans:

@Component

**public class** MyBean {

**private final** KafkaTemplate kafkaTemplate; @Autowired

**public** MyBean(KafkaTemplate kafkaTemplate) {

**this**.kafkaTemplate = kafkaTemplate;

}

*// ...*

}

### Receiving a Message

When the Apache Kafka infrastructure is present, any bean can be annotated with @KafkaListener to create a listener endpoint. If no KafkaListenerContainerFactory has been defined, a default one is configured automatically with keys defined in spring.kafka.listener.\*.

The following component creates a listener endpoint on the someTopic topic:

@Component

**public class** MyBean {

@KafkaListener(topics = "someTopic")

**public void** processMessage(String content) {

*// ...*

}

}

### Additional Kafka Properties

The properties supported by auto configuration are shown in [Appendix A, *Common application*](#_bookmark569)[*properties*](#_bookmark569). Note that these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties for the most part, refer to the Apache Kafka documentation for details.

The first few of these properties apply to both producers and consumers, but can be specified at the producer or consumer level if you wish to use different values for each. Apache Kafka designates properties with an importance: HIGH, MEDIUM and LOW. Spring Boot auto configuration supports all HIGH importance properties, some selected MEDIUM and LOW, and any that do not have a default value.

Only a subset of the properties supported by Kafka are available via the KafkaProperties class. If you wish to configure the producer or consumer with additional properties that are not directly supported, use the following:

**spring.kafka.properties.foo.bar**=baz

This sets the common foo.bar Kafka property to baz.

These properties will be shared by both the consumer and producer factory beans. If you wish to customize these components with different properties, such as to use a different metrics reader for each, you can override the bean definitions, as follows:

@Configuration

**public static class** CustomKafkaBeans {

**/\*\***

* **Customized ProducerFactory bean.**
* **@param properties the kafka properties.**

**\* @return the bean.**

**\*/**

@Bean

**public** ProducerFactory<?, ?> kafkaProducerFactory(KafkaProperties properties) { Map<String, Object> producerProperties = properties.buildProducerProperties(); producerProperties.put(CommonClientConfigs.METRIC\_REPORTER\_CLASSES\_CONFIG,

MyProducerMetricsReporter.**class**);

**return new** DefaultKafkaProducerFactory<Object, Object>(producerProperties);

}

**/\*\***

* **Customized ConsumerFactory bean.**
* **@param properties the kafka properties.**
* **@return the bean.**

**\*/**

@Bean

**public** ConsumerFactory<?, ?> kafkaConsumerFactory(KafkaProperties properties) { Map<String, Object> consumerProperties = properties.buildConsumerProperties(); consumerProperties.put(CommonClientConfigs.METRIC\_REPORTER\_CLASSES\_CONFIG,

MyConsumerMetricsReporter.**class**);

**return new** DefaultKafkaConsumerFactory<Object, Object>(consumerProperties);

}

}

# Calling REST services

If you need to call remote REST services from your application, you can use Spring Framework’s RestTemplate class. Since RestTemplate instances often need to be customized before being used, Spring Boot does not provide any single auto-configured RestTemplate bean. It does, however, auto-configure a RestTemplateBuilder which can be used to create RestTemplate instances when needed. The auto-configured RestTemplateBuilder will ensure that sensible HttpMessageConverters are applied to RestTemplate instances.

Here’s a typical example:

@Service

**public class** MyBean {

**private final** RestTemplate restTemplate;

**public** MyBean(RestTemplateBuilder restTemplateBuilder) {

**this**.restTemplate = restTemplateBuilder.build();

}

**public** Details someRestCall(String name) {

**return this**.restTemplate.getForObject(***"/{name}/details"***, Details.**class**, name);

}

}

**Tip**

RestTemplateBuilder includes a number of useful methods that can be used to quickly configure a RestTemplate. For example, to add BASIC auth support you can use builder.basicAuthorization("user", "password").build().

## RestTemplate customization

There are three main approaches to RestTemplate customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured RestTemplateBuilder and then call its methods as required. Each method call returns a new RestTemplateBuilder instance so the customizations will only affect this use of the builder.

To make an application-wide, additive customization a RestTemplateCustomizer bean can be used. All such beans are automatically registered with the auto-configured RestTemplateBuilder and will be applied to any templates that are built with it.

Here’s an example of a customizer that configures the use of a proxy for all hosts except 192.168.0.5:

**static class** ProxyCustomizer **implements** RestTemplateCustomizer {

@Override

**public void** customize(RestTemplate restTemplate) { HttpHost proxy = **new** HttpHost(***"proxy.example.com"***); HttpClient httpClient = HttpClientBuilder.create()

.setRoutePlanner(**new** DefaultProxyRoutePlanner(proxy) {

@Override

**public** HttpHost determineProxy(HttpHost target, HttpRequest request, HttpContext context)

**throws** HttpException {

**if** (target.getHostName().equals(***"192.168.0.5"***)) {

**return** null;

}

**return super**.determineProxy(target, request, context);

}

}).build(); restTemplate.setRequestFactory(

**new** HttpComponentsClientHttpRequestFactory(httpClient));

}

}

Lastly, the most extreme (and rarely used) option is to create your own RestTemplateBuilder bean. This will switch off the auto-configuration of a RestTemplateBuilder and will prevent any RestTemplateCustomizer beans from being used.

# Validation

The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (e.g. Hibernate validator) is on the classpath. This allows bean methods to be annotated with javax.validation constraints on their parameters and/or on their return value. Target classes with such annotated methods need to be annotated with the @Validated annotation at the type level for their methods to be searched for inline constraint annotations.

For instance, the following service triggers the validation of the first argument, making sure its size is between 8 and 10

@Service @Validated

**public class** MyBean {

**public** Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code, Author author) {

...

}

}

# Sending email

The Spring Framework provides an easy abstraction for sending email using the JavaMailSender

interface and Spring Boot provides auto-configuration for it as well as a starter module.

**Tip**

Check the [reference documentation](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#mail) for a detailed explanation of how you can use

JavaMailSender.

If spring.mail.host and the relevant libraries (as defined by spring-boot-starter-mail) are available, a default JavaMailSender is created if none exists. The sender can be further customized by configuration items from the spring.mail namespace, see the [MailProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mail/MailProperties.java) for more details.

In particular, certain default timeout values are infinite and you may want to change that to avoid having a thread blocked by an unresponsive mail server:

**spring.mail.properties.mail.smtp.connectiontimeout**=5000 **spring.mail.properties.mail.smtp.timeout**=3000 **spring.mail.properties.mail.smtp.writetimeout**=5000

# Distributed Transactions with JTA

Spring Boot supports distributed JTA transactions across multiple XA resources using either an [Atomikos](http://www.atomikos.com/) or [Bitronix](https://github.com/bitronix/btm) embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

When a JTA environment is detected, Spring’s JtaTransactionManager will be used to manage transactions. Auto-configured JMS, DataSource and JPA beans will be upgraded to support XA transactions. You can use standard Spring idioms such as @Transactional to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions you can set the spring.jta.enabled property to false to disable the JTA auto-configuration.

## Using an Atomikos transaction manager

Atomikos is a popular open source transaction manager which can be embedded into your Spring Boot application. You can use the spring-boot-starter-jta-atomikos Starter to pull in the appropriate Atomikos libraries. Spring Boot will auto-configure Atomikos and ensure that appropriate depends-on settings are applied to your Spring beans for correct startup and shutdown ordering.

By default Atomikos transaction logs will be written to a transaction-logs directory in your application home directory (the directory in which your application jar file resides). You can customize this directory by setting a spring.jta.log-dir property in your application.properties file. Properties starting spring.jta.atomikos.properties can also be used to customize the Atomikos UserTransactionServiceImp. See the [AtomikosProperties Javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/jta/atomikos/AtomikosProperties.html) for complete details.

**Note**

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Atomikos instance must be configured with a unique ID. By default this ID is the IP address of the machine on which Atomikos is running. To ensure uniqueness in production, you should configure the spring.jta.transaction-manager-id property with a different value for each instance of your application.

## Using a Bitronix transaction manager

Bitronix is popular open source JTA transaction manager implementation. You can use the spring- boot-starter-jta-bitronix starter to add the appropriate Bitronix dependencies to your project. As with Atomikos, Spring Boot will automatically configure Bitronix and post-process your beans to ensure that startup and shutdown ordering is correct.

By default Bitronix transaction log files (part1.btm and part2.btm) will be written to a transaction-logs directory in your application home directory. You can customize this directory by using the spring.jta.log-dir property. Properties starting spring.jta.bitronix.properties are also bound to the bitronix.tm.Configuration bean, allowing for complete customization. See the [Bitronix documentation](https://github.com/bitronix/btm/wiki/Transaction-manager-configuration) for details.

**Note**

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Bitronix instance must be configured with a unique ID. By default this ID is the IP address

of the machine on which Bitronix is running. To ensure uniqueness in production, you should configure the spring.jta.transaction-manager-id property with a different value for each instance of your application.

## Using a Narayana transaction manager

Narayana is popular open source JTA transaction manager implementation supported by JBoss. You can use the spring-boot-starter-jta-narayana starter to add the appropriate Narayana dependencies to your project. As with Atomikos and Bitronix, Spring Boot will automatically configure Narayana and post-process your beans to ensure that startup and shutdown ordering is correct.

By default Narayana transaction logs will be written to a transaction-logs directory in your application home directory (the directory in which your application jar file resides). You can customize this directory by setting a spring.jta.log-dir property in your application.properties file. Properties starting spring.jta.narayana.properties can also be used to customize the Narayana configuration. See the [NarayanaProperties Javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/jta/narayana/NarayanaProperties.html) for complete details.

**Note**

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Narayana instance must be configured with a unique ID. By default this ID is set to 1. To ensure uniqueness in production, you should configure the spring.jta.transaction- manager-id property with a different value for each instance of your application.

## Using a Java EE managed transaction manager

If you are packaging your Spring Boot application as a war or ear file and deploying it to a Java EE application server, you can use your application servers built-in transaction manager. Spring Boot will attempt to auto-configure a transaction manager by looking at common JNDI locations (java:comp/UserTransaction, java:comp/TransactionManager etc). If you are using a transaction service provided by your application server, you will generally also want to ensure that all resources are managed by the server and exposed over JNDI. Spring Boot will attempt to auto-configure JMS by looking for a ConnectionFactory at the JNDI path java:/JmsXA or java:/XAConnectionFactory and you can use the [spring.datasource.jndi-name property](#_bookmark169) to configure your DataSource.

## Mixing XA and non-XA JMS connections

When using JTA, the primary JMS ConnectionFactory bean will be XA aware and participate in distributed transactions. In some situations you might want to process certain JMS messages using a non-XA ConnectionFactory. For example, your JMS processing logic might take longer than the XA timeout.

If you want to use a non-XA ConnectionFactory you can inject the nonXaJmsConnectionFactory bean rather than the @Primary jmsConnectionFactory bean. For consistency the jmsConnectionFactory bean is also provided using the bean alias xaJmsConnectionFactory.

For example:

*// Inject the primary (XA aware) ConnectionFactory*

@Autowired

**private** ConnectionFactory defaultConnectionFactory;

*// Inject the XA aware ConnectionFactory (uses the alias and injects the same as above)*

@Autowired @Qualifier("xaJmsConnectionFactory")

**private** ConnectionFactory xaConnectionFactory;

*// Inject the non-XA aware ConnectionFactory* @Autowired @Qualifier("nonXaJmsConnectionFactory")

**private** ConnectionFactory nonXaConnectionFactory;

## Supporting an alternative embedded transaction manager

The [XAConnectionFactoryWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/XAConnectionFactoryWrapper.java) and [XADataSourceWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/XADataSourceWrapper.java) interfaces can be used to support alternative embedded transaction managers. The interfaces are responsible for wrapping XAConnectionFactory and XADataSource beans and exposing them as regular ConnectionFactory and DataSource beans which will transparently enroll in the distributed transaction. DataSource and JMS auto-configuration will use JTA variants as long as you have a JtaTransactionManager bean and appropriate XA wrapper beans registered within your ApplicationContext.

The [BitronixXAConnectionFactoryWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/bitronix/BitronixXAConnectionFactoryWrapper.java) and [BitronixXADataSourceWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/bitronix/BitronixXADataSourceWrapper.java) provide good examples of how to write XA wrappers.

# Hazelcast

If Hazelcast is on the classpath, Spring Boot will auto-configure a HazelcastInstance that you can inject in your application. The HazelcastInstance is only created if a configuration is found.

You can define a com.hazelcast.config.Config bean and we’ll use that. If your configuration defines an instance name, we’ll try to locate an existing instance rather than creating a new one.

You could also specify the hazelcast.xml configuration file to use via configuration:

**spring.hazelcast.config**=classpath:config/my-hazelcast.xml

Otherwise, Spring Boot tries to find the Hazelcast configuration from the default locations, that is hazelcast.xml in the working directory or at the root of the classpath. We also check if the hazelcast.config system property is set. Check the [Hazelcast documentation](http://docs.hazelcast.org/docs/latest/manual/html-single/) for more details.

**Note**

Spring Boot also has an [explicit caching support for Hazelcast](#_bookmark220). The HazelcastInstance is automatically wrapped in a CacheManager implementation if caching is enabled.

# Spring Integration

Spring Boot offers several conveniences for working with Spring Integration, including the spring- boot-starter-integration ‘Starter’. Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP etc. If Spring Integration is available on your classpath it will be initialized through the @EnableIntegration annotation. Message processing statistics will be published over JMX if 'spring-integration-jmx' is also on the classpath. See the [IntegrationAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/integration/IntegrationAutoConfiguration.java) class for more details.

# Spring Session

Spring Boot provides Spring Session auto-configuration for a wide range of stores:

* JDBC
* MongoDB
* Redis
* Hazelcast
* HashMap

If Spring Session is available, you must choose the [StoreType](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/session/StoreType.java) that you wish to use to store the sessions. For instance to use JDBC as backend store, you’d configure your application as follows:

**spring.session.store-type**=jdbc

**Tip**

You can disable Spring Session by setting the store-type to none.

Each store has specific additional settings. For instance it is possible to customize the name of the table for the jdbc store:

**spring.session.jdbc.table-name**=SESSIONS

# Monitoring and management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default Spring Boot will create an MBeanServer with bean id ‘mbeanServer’ and expose any of your beans that are annotated with Spring JMX annotations (@ManagedResource, @ManagedAttribute, @ManagedOperation).

See the [JmxAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jmx/JmxAutoConfiguration.java) class for more details.

# Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules; spring-boot-test contains core items, and spring-boot- test-autoconfigure supports auto-configuration for tests.

Most developers will just use the spring-boot-starter-test ‘Starter’ which imports both Spring Boot test modules as well has JUnit, AssertJ, Hamcrest and a number of other useful libraries.

## Test scope dependencies

If you use the spring-boot-starter-test ‘Starter’ (in the test scope), you will find the following provided libraries:

* [JUnit](http://junit.org/) — The de-facto standard for unit testing Java applications.
* [Spring Test](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#integration-testing) & Spring Boot Test — Utilities and integration test support for Spring Boot applications.
* [AssertJ](http://joel-costigliola.github.io/assertj/) — A fluent assertion library.
* [Hamcrest](http://hamcrest.org/JavaHamcrest/) — A library of matcher objects (also known as constraints or predicates).
* [Mockito](http://mockito.org/) — A Java mocking framework.
* [JSONassert](https://github.com/skyscreamer/JSONassert) — An assertion library for JSON.
* [JsonPath](https://github.com/jayway/JsonPath) — XPath for JSON.

**Note**

By default, Spring Boot uses Mockito 1.x. However it’s also possible to use 2.x if you wish.

These are common libraries that we generally find useful when writing tests. You are free to add additional test dependencies of your own if these don’t suit your needs.

## Testing Spring applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can simply instantiate objects using the new operator without even involving Spring. You can also use *mock objects* instead of real dependencies.

Often you need to move beyond ‘unit testing’ and start ‘integration testing’ (with a Spring ApplicationContext actually involved in the process). It’s useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

The Spring Framework includes a dedicated test module for just such integration testing. You can declare a dependency directly to org.springframework:spring-test or use the spring-boot- starter-test ‘Starter’ to pull it in transitively.

If you have not used the spring-test module before you should start by reading the [relevant section](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#testing) of the Spring Framework reference documentation.

## Testing Spring Boot applications

A Spring Boot application is just a Spring ApplicationContext, so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context. One thing to watch out for though is that the external properties, logging and other features of Spring Boot are only installed in the context by default if you use SpringApplication to create it.

Spring Boot provides a @SpringBootTest annotation which can be used as an alternative to the standard spring-test @ContextConfiguration annotation when you need Spring Boot features. The annotation works by creating the ApplicationContext used in your tests via SpringApplication.

You can use the webEnvironment attribute of @SpringBootTest to further refine how your tests will run:

* MOCK — Loads a WebApplicationContext and provides a mock servlet environment. Embedded servlet containers are not started when using this annotation. If servlet APIs are not on your classpath this mode will transparently fallback to creating a regular non-web ApplicationContext. Can be used in conjunction with @AutoConfigureMockMvc for MockMvc-based testing of your application.
* RANDOM\_PORT — Loads an EmbeddedWebApplicationContext and provides a real servlet environment. Embedded servlet containers are started and listening on a random port.
* DEFINED\_PORT — Loads an EmbeddedWebApplicationContext and provides a real servlet environment. Embedded servlet containers are started and listening on a defined port (i.e from your application.properties or on the default port 8080).
* NONE — Loads an ApplicationContext using SpringApplication but does not provide *any*

servlet environment (mock or otherwise).

**Note**

If your test is @Transactional, it will rollback the transaction at the end of each test method by default. However, as using this arrangement with either RANDOM\_PORT or DEFINED\_PORT implicitly provides a real servlet environment, HTTP client and server will run in separate threads, thus separate transactions. Any transaction initiated on the server won’t rollback in this case.

**Note**

In addition to @SpringBootTest a number of other annotations are also provided for testing more specific slices of an application. See below for details.

**Tip**

Don’t forget to also add @RunWith(SpringRunner.class) to your test, otherwise the annotations will be ignored.

### Detecting test configuration

If you’re familiar with the Spring Test Framework, you may be used to using @ContextConfiguration(classes=…) in order to specify which Spring @Configuration to load. Alternatively, you might have often used nested @Configuration classes within your test.

When testing Spring Boot applications this is often not required. Spring Boot’s @\*Test annotations will search for your primary configuration automatically whenever you don’t explicitly define one.

The search algorithm works up from the package that contains the test until it finds a @SpringBootApplication or @SpringBootConfiguration annotated class. As long as you’ve [structured your code](#_bookmark50) in a sensible way your main configuration is usually found.

If you want to customize the primary configuration, you can use a nested @TestConfiguration class. Unlike a nested @Configuration class which would be used instead of a your application’s primary configuration, a nested @TestConfiguration class will be used in addition to your application’s primary configuration.

**Note**

Spring’s test framework will cache application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it’s discovered), the potentially time consuming process of loading the context will only happen once.

### Excluding test configuration

If your application uses component scanning, for example if you use @SpringBootApplication or @ComponentScan, you may find top-level configuration classes created only for specific tests accidentally get picked up everywhere.

As we [have seen above](#_bookmark262), @TestConfiguration can be used on an inner class of a test to customize the primary configuration. When placed on a top-level class, @TestConfiguration indicates that classes in src/test/java should not be picked up by scanning. You can then import that class explicitly where it is required:

@RunWith(SpringRunner.class) @SpringBootTest @Import(MyTestsConfiguration.class) **public class** MyTests {

@Test

**public void** exampleTest() {

...

}

}

**Note**

If you directly use @ComponentScan (i.e. not via @SpringBootApplication) you will need to register the TypeExcludeFilter with it. See [the Javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/context/TypeExcludeFilter.html) for details.

### Working with random ports

If you need to start a full running server for tests, we recommend that you use random ports. If you use @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT) an available port will be picked at random each time your test runs.

The @LocalServerPort annotation can be used to [inject the actual port used](#_bookmark463) into your test. For convenience, tests that need to make REST calls to the started server can additionally @Autowire a TestRestTemplate which will resolve relative links to the running server.

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

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

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.boot.test.context.SpringBootTest.WebEnvironment;

**import** org.springframework.boot.test.web.client.TestRestTemplate;

**import** org.springframework.test.context.junit4.SpringRunner;

**import static** org.assertj.core.api.Assertions.assertThat; @RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM\_PORT)

**public class** RandomPortExampleTests {

@Autowired

**private** TestRestTemplate restTemplate;

@Test

**public void** exampleTest() {

String body = **this**.restTemplate.getForObject(***"/"***, String.**class**); assertThat(body).isEqualTo(***"Hello World"***);

}

}

### Mocking and spying beans

It’s sometimes necessary to mock certain components within your application context when running tests. For example, you may have a facade over some remote service that’s unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a @MockBean annotation that can be used to define a Mockito mock for a bean inside your ApplicationContext. You can use the annotation to add new beans, or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test, or on @Configuration classes and fields. When used on a field, the instance of the created mock will also be injected. Mock beans are automatically reset after each test method.

**Note**

This feature is automatically enabled as long as your test uses one of Spring Boot’s test annotations (i.e. @SpringBootTest). To use this feature with a different arrangement, a listener will need to be added explicitly:

@TestExecutionListeners(MockitoTestExecutionListener.**class**)

Here’s a typical example where we replace an existing RemoteService bean with a mock implementation:

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.beans.factory.annotation.\*;

**import** org.springframework.boot.test.context.\*; **import** org.springframework.boot.test.mock.mockito.\*; **import** org.springframework.test.context.junit4.\*;

**import static** org.assertj.core.api.Assertions.\*;

**import static** org.mockito.BDDMockito.\*;

@RunWith(SpringRunner.class) @SpringBootTest

**public class** MyTests {

@MockBean

**private** RemoteService remoteService;

@Autowired

**private** Reverser reverser;

@Test

**public void** exampleTest() {

*// RemoteService has been injected into the reverser bean* given(**this**.remoteService.someCall()).willReturn(***"mock"***); String reverse = reverser.reverseSomeCall(); assertThat(reverse).isEqualTo(***"kcom"***);

}

}

Additionally you can also use @SpyBean to wrap any existing bean with a Mockito spy. See the Javadoc for full details.

### Auto-configured tests

Spring Boot’s auto-configuration system works well for applications, but can sometimes be a little too much for tests. It’s often helpful to load only the parts of the configuration that are required to test a ‘slice’ of your application. For example, you might want to test that Spring MVC controllers are mapping URLs correctly, and you don’t want to involve database calls in those tests; or you *might be wanting* to test JPA entities, and you’re not interested in web layer when those tests run.

The spring-boot-test-autoconfigure module includes a number of annotations that can be used to automatically configure such ‘slices’. Each of them works in a similar way, providing a @…Test annotation that loads the ApplicationContext and one or more @AutoConfigure… annotations that can be used to customize auto-configuration settings.

**Note**

Each slice loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most @…Test annotations provide an excludeAutoConfiguration attribute. Alternatively, you can use @ImportAutoConfiguration#exclude.

**Tip**

It’s also possible to use the @AutoConfigure… annotations with the standard @SpringBootTest annotation. You can use this combination if you’re not interested in ‘slicing’ your application but you want some of the auto-configured test beans.

### Auto-configured JSON tests

To test that Object JSON serialization and deserialization is working as expected you can use the @JsonTest annotation. @JsonTest will auto-configure Jackson ObjectMapper, any @JsonComponent beans and any Jackson Modules. It also configures Gson if you happen to be using that instead of, or as well as, Jackson. If you need to configure elements of the auto-configuration you can use the @AutoConfigureJsonTesters annotation.

Spring Boot includes AssertJ based helpers that work with the JSONassert and JsonPath libraries to check that JSON is as expected. The JacksonTester, GsonTester and BasicJsonTester classes

can be used for Jackson, Gson and Strings respectively. Any helper fields on the test class can be

@Autowired when using @JsonTest.

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.beans.factory.annotation.\*; **import** org.springframework.boot.test.autoconfigure.json.\*; **import** org.springframework.boot.test.context.\*;

**import** org.springframework.boot.test.json.\*;

**import** org.springframework.test.context.junit4.\*;

**import static** org.assertj.core.api.Assertions.\*; @RunWith(SpringRunner.class)

@JsonTest

**public class** MyJsonTests {

@Autowired

**private** JacksonTester<VehicleDetails> json;

@Test

**public void** testSerialize() **throws** Exception {

VehicleDetails details = **new** VehicleDetails(***"Honda"***, ***"Civic"***);

*// Assert against a `.json` file in the same package as the test*

assertThat(**this**.json.write(details)).isEqualToJson(***"expected.json"***);

*// Or use JSON path based assertions* assertThat(**this**.json.write(details)).hasJsonPathStringValue(***"@.make"***); assertThat(**this**.json.write(details)).extractingJsonPathStringValue(***"@.make"***)

.isEqualTo(***"Honda"***);

}

@Test

**public void** testDeserialize() **throws** Exception {

String content = ***"{\"make\":\"Ford\",\"model\":\"Focus\"}"***; assertThat(**this**.json.parse(content))

.isEqualTo(**new** VehicleDetails(***"Ford"***, ***"Focus"***)); assertThat(**this**.json.parseObject(content).getMake()).isEqualTo(***"Ford"***);

}

}

**Note**

JSON helper classes can also be used directly in standard unit tests. Simply call the initFields

method of the helper in your @Before method if you aren’t using @JsonTest.

A list of the auto-configuration that is enabled by @JsonTest can be [found in the appendix](#_bookmark592).

### Auto-configured Spring MVC tests

To test Spring MVC controllers are working as expected you can use the @WebMvcTest annotation. @WebMvcTest will auto-configure the Spring MVC infrastructure and limit scanned beans to @Controller, @ControllerAdvice, @JsonComponent, Filter, WebMvcConfigurer and HandlerMethodArgumentResolver. Regular @Component beans will not be scanned when using this annotation.

Often @WebMvcTest will be limited to a single controller and used in combination with @MockBean to provide mock implementations for required collaborators.

@WebMvcTest also auto-configures MockMvc. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.

**Tip**

You can also auto-configure MockMvc in a non-@WebMvcTest (e.g. SpringBootTest) by annotating it with @AutoConfigureMockMvc.

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.beans.factory.annotation.\*;

**import** org.springframework.boot.test.autoconfigure.web.servlet.\*;

**import** org.springframework.boot.test.mock.mockito.\*;

**import static** org.assertj.core.api.Assertions.\*;

**import static** org.mockito.BDDMockito.\*;

**import static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

**import static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@RunWith(SpringRunner.class) @WebMvcTest(UserVehicleController.class) **public class** MyControllerTests {

@Autowired

**private** MockMvc mvc;

@MockBean

**private** UserVehicleService userVehicleService;

@Test

**public void** testExample() **throws** Exception { given(**this**.userVehicleService.getVehicleDetails(***"sboot"***))

.willReturn(**new** VehicleDetails(***"Honda"***, ***"Civic"***));

**this**.mvc.perform(get(***"/sboot/vehicle"***).accept(MediaType.TEXT\_PLAIN))

.andExpect(status().isOk()).andExpect(content().string(***"Honda Civic"***));

}

}

**Tip**

If you need to configure elements of the auto-configuration (for example when servlet filters should be applied) you can use attributes in the @AutoConfigureMockMvc annotation.

If you use HtmlUnit or Selenium, auto-configuration will also provide a WebClient bean and/or a

WebDriver bean. Here is an example that uses HtmlUnit:

**import** com.gargoylesoftware.htmlunit.\*;

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.beans.factory.annotation.\*;

**import** org.springframework.boot.test.autoconfigure.web.servlet.\*;

**import** org.springframework.boot.test.mock.mockito.\*;

**import static** org.assertj.core.api.Assertions.\*;

**import static** org.mockito.BDDMockito.\*;

@RunWith(SpringRunner.class) @WebMvcTest(UserVehicleController.class) **public class** MyHtmlUnitTests {

@Autowired

**private** WebClient webClient;

@MockBean

**private** UserVehicleService userVehicleService;

@Test

**public void** testExample() **throws** Exception { given(**this**.userVehicleService.getVehicleDetails(***"sboot"***))

.willReturn(**new** VehicleDetails(***"Honda"***, ***"Civic"***)); HtmlPage page = **this**.webClient.getPage(***"/sboot/vehicle.html"***);

assertThat(page.getBody().getTextContent()).isEqualTo(***"Honda Civic"***);

}

}

**Note**

By default Spring Boot will put WebDriver beans in a special “scope” to ensure that the driver is quit after each test, and that a new instance is injected. If you don’t want this behavior you can add @Scope("singleton") to your WebDriver @Bean definition.

A list of the auto-configuration that is enabled by @WebMvcTest can be [found in the appendix](#_bookmark592).

### Auto-configured Data JPA tests

@DataJpaTest can be used if you want to test JPA applications. By default it will configure an in- memory embedded database, scan for @Entity classes and configure Spring Data JPA repositories. Regular @Component beans will not be loaded into the ApplicationContext.

Data JPA tests are transactional and rollback at the end of each test by default, see the [relevant section](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle#testcontext-tx-enabling-transactions) in the Spring Reference Documentation for more details. If that’s not what you want, you can disable transaction management for a test or for the whole class as follows:

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

**import** org.springframework.test.context.junit4.SpringRunner; **import** org.springframework.transaction.annotation.Propagation; **import** org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class) @DataJpaTest

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**public class** ExampleNonTransactionalTests {

}

Data JPA tests may also inject a [TestEntityManager](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-test-autoconfigure/src/main/java/org/springframework/boot/test/autoconfigure/orm/jpa/TestEntityManager.java) bean which provides an alternative to the standard JPA EntityManager specifically designed for tests. If you want to use TestEntityManager outside of @DataJpaTests you can also use the @AutoConfigureTestEntityManager annotation. A JdbcTemplate is also available if you need that.

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.boot.test.autoconfigure.orm.jpa.\*;

**import static** org.assertj.core.api.Assertions.\*; @RunWith(SpringRunner.class)

@DataJpaTest

**public class** ExampleRepositoryTests {

@Autowired

**private** TestEntityManager entityManager;

@Autowired

**private** UserRepository repository;

@Test

**public void** testExample() **throws** Exception { **this**.entityManager.persist(**new** User(***"sboot"***, ***"1234"***)); User user = **this**.repository.findByUsername(***"sboot"***); assertThat(user.getUsername()).isEqualTo(***"sboot"***); assertThat(user.getVin()).isEqualTo(***"1234"***);

}

}

In-memory embedded databases generally work well for tests since they are fast and don’t require any developer installation. If, however, you prefer to run tests against a real database you can use the @AutoConfigureTestDatabase annotation:

@RunWith(SpringRunner.class) @DataJpaTest

@AutoConfigureTestDatabase(replace=Replace.NONE)

**public class** ExampleRepositoryTests {

*// ...*

}

A list of the auto-configuration that is enabled by @DataJpaTest can be [found in the appendix](#_bookmark592).

### Auto-configured JDBC tests

@JdbcTest is similar to @DataJpaTest but for pure jdbc-related tests. By default it will also configure an in-memory embedded database and a JdbcTemplate. Regular @Component beans will not be loaded into the ApplicationContext.

JDBC tests are transactional and rollback at the end of each test by default, see the [relevant section](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle#testcontext-tx-enabling-transactions) in the Spring Reference Documentation for more details. If that’s not what you want, you can disable transaction management for a test or for the whole class as follows:

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.boot.test.autoconfigure.jdbc.JdbcTest; **import** org.springframework.test.context.junit4.SpringRunner; **import** org.springframework.transaction.annotation.Propagation; **import** org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class) @JdbcTest

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**public class** ExampleNonTransactionalTests {

}

If you prefer your test to run against a real database, you can use the @AutoConfigureTestDatabase

annotation the same way as for DataJpaTest.

A list of the auto-configuration that is enabled by @JdbcTest can be [found in the appendix](#_bookmark592).

### Auto-configured Data MongoDB tests

@DataMongoTest can be used if you want to test MongoDB applications. By default, it will configure an in-memory embedded MongoDB (if available), configure a MongoTemplate, scan for @Document classes and configure Spring Data MongoDB repositories. Regular @Component beans will not be loaded into the ApplicationContext:

**import** org.junit.runner.RunWith;

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

**import** org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;

**import** org.springframework.data.mongodb.core.MongoTemplate;

**import** org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class) @DataMongoTest

**public class** ExampleDataMongoTests {

@Autowired

**private** MongoTemplate mongoTemplate;

*//*

}

In-memory embedded MongoDB generally works well for tests since it is fast and doesn’t require any developer installation. If, however, you prefer to run tests against a real MongoDB server you should exclude the embedded MongoDB auto-configuration:

**import** org.junit.runner.RunWith;

**import** org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;

**import** org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;

**import** org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)

@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)

**public class** ExampleDataMongoNonEmbeddedTests {

}

A list of the auto-configuration that is enabled by @DataMongoTest can be [found in the appendix](#_bookmark592).

### Auto-configured REST clients

The @RestClientTest annotation can be used if you want to test REST clients. By default it will auto-configure Jackson and GSON support, configure a RestTemplateBuilder and add support for MockRestServiceServer. The specific beans that you want to test should be specified using value or components attribute of @RestClientTest:

@RunWith(SpringRunner.class) @RestClientTest(RemoteVehicleDetailsService.class) **public class** ExampleRestClientTest {

@Autowired

**private** RemoteVehicleDetailsService service;

@Autowired

**private** MockRestServiceServer server;

@Test

**public void** getVehicleDetailsWhenResultIsSuccessShouldReturnDetails()

**throws** Exception {

**this**.server.expect(requestTo(***"/greet/details"***))

.andRespond(withSuccess(***"hello"***, MediaType.TEXT\_PLAIN)); String greeting = **this**.service.callRestService(); assertThat(greeting).isEqualTo(***"hello"***);

}

}

A list of the auto-configuration that is enabled by @RestClientTest can be [found in the appendix](#_bookmark592).

### Auto-configured Spring REST Docs tests

The @AutoConfigureRestDocs annotation can be used if you want to use Spring REST Docs in your tests. It will automatically configure MockMvc to use Spring REST Docs and remove the need for Spring REST Docs' JUnit rule.

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

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

**import** org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

**import** org.springframework.http.MediaType;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** org.springframework.test.web.servlet.MockMvc;

**import static** org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document; **import static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get; **import static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@RunWith(SpringRunner.class) @WebMvcTest(UserController.class) @AutoConfigureRestDocs("target/generated-snippets") **public class** UserDocumentationTests {

@Autowired

**private** MockMvc mvc;

@Test

**public void** listUsers() **throws** Exception {

**this**.mvc.perform(get(***"/users"***).accept(MediaType.TEXT\_PLAIN))

.andExpect(status().isOk())

.andDo(document(***"list-users"***));

}

}

In addition to configuring the output directory, @AutoConfigureRestDocs can also configure the host, scheme, and port that will appear in any documented URIs. If you require more control over Spring REST Docs' configuration a RestDocsMockMvcConfigurationCustomizer bean can be used:

@TestConfiguration

**static class** CustomizationConfiguration

**implements** RestDocsMockMvcConfigurationCustomizer {

@Override

**public void** customize(MockMvcRestDocumentationConfigurer configurer) { configurer.snippets().withTemplateFormat(TemplateFormats.markdown());

}

}

If you want to make use of Spring REST Docs' support for a parameterized output directory, you can create a RestDocumentationResultHandler bean. The auto-configuration will call alwaysDo with this result handler, thereby causing each MockMvc call to automatically generate the default snippets:

@TestConfiguration

**static class** ResultHandlerConfiguration {

@Bean

**public** RestDocumentationResultHandler restDocumentation() {

**return** MockMvcRestDocumentation.document(***"{method-name}"***);

}

}

### Using Spock to test Spring Boot applications

If you wish to use Spock to test a Spring Boot application you should add a dependency on Spock’s spock-spring module to your application’s build. spock-spring integrates Spring’s test framework into Spock. Exactly how you can use Spock to test a Spring Boot application depends on the version of Spock that you are using.

**Note**

Spring Boot provides dependency management for Spock 1.0. If you wish to use Spock 1.1 you should [override the spock.version property](#_bookmark550) in your build.gradle or pom.xml file.

When using Spock 1.1, the annotations [described above](#_bookmark261) can only be used and you can annotate your

Specification with @SpringBootTest to suit the needs of your tests.

When using Spock 1.0, @SpringBootTest will not work for a web project. You need to use @SpringApplicationConfiguration and @WebIntegrationTest(randomPort = true). Being unable to use @SpringBootTest means that you also lose the auto-configured TestRestTemplate bean. You can create an equivalent bean yourself using the following configuration:

@Configuration

**static class** TestRestTemplateConfiguration {

@Bean

**public** TestRestTemplate testRestTemplate( ObjectProvider<RestTemplateBuilder> builderProvider, Environment environment) {

RestTemplateBuilder builder = builderProvider.getIfAvailable(); TestRestTemplate template = builder == null ? **new** TestRestTemplate()

: **new** TestRestTemplate(builder.build()); template.setUriTemplateHandler(**new** LocalHostUriTemplateHandler(environment)); **return** template;

}

}

## Test utilities

A few test utility classes are packaged as part of spring-boot that are generally useful when testing your application.

### ConfigFileApplicationContextInitializer

ConfigFileApplicationContextInitializer is an ApplicationContextInitializer that can apply to your tests to load Spring Boot application.properties files. You can use this when you don’t need the full features provided by @SpringBootTest.

@ContextConfiguration(classes = Config.**class**,

initializers = ConfigFileApplicationContextInitializer.**class**)

**Note**

Using ConfigFileApplicationContextInitializer alone won’t provide support for @Value("${…}") injection. Its only job is to ensure that application.properties files are loaded into Spring’s Environment. For @Value support you need to either additionally configure

a PropertySourcesPlaceholderConfigurer or use @SpringBootTest where one will be auto-configured for you.

### EnvironmentTestUtils

EnvironmentTestUtils allows you to quickly add properties to a ConfigurableEnvironment or

ConfigurableApplicationContext. Simply call it with key=value strings:

EnvironmentTestUtils.addEnvironment(env, ***"org=Spring"***, ***"name=Boot"***);

### OutputCapture

OutputCapture is a JUnit Rule that you can use to capture System.out and System.err output. Simply declare the capture as a @Rule then use toString() for assertions:

**import** org.junit.Rule;

**import** org.junit.Test;

**import** org.springframework.boot.test.rule.OutputCapture;

**import static** org.hamcrest.Matchers.\*;

**import static** org.junit.Assert.\*;

**public class** MyTest { @Rule

**public** OutputCapture capture = **new** OutputCapture();

@Test

**public void** testName() **throws** Exception { System.out.println(***"Hello World!"***); assertThat(capture.toString(), containsString(***"World"***));

}

}

### TestRestTemplate

TestRestTemplate is a convenience alternative to Spring’s RestTemplate that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case the template will behave in a test-friendly way by not throwing exceptions on server-side errors. It is recommended, but not mandatory, to use Apache HTTP Client (version 4.3.2 or better), and if you have that on your classpath the TestRestTemplate will respond by configuring the client appropriately. If you do use Apache’s HTTP client some additional test-friendly features will be enabled:

* Redirects will not be followed (so you can assert the response location)
* Cookies will be ignored (so the template is stateless)

TestRestTemplate can be instantiated directly in your integration tests:

**public class** MyTest {

**private** TestRestTemplate template = **new** TestRestTemplate(); @Test

**public void** testRequest() **throws** Exception {

HttpHeaders headers = template.getForEntity([***"http://myhost.com/example"***](http://myhost.com/example), String.**class**).getHeaders();

assertThat(headers.getLocation().toString(), containsString(***"myotherhost"***));

}

}

Alternatively, if you are using the @SpringBootTest annotation with WebEnvironment.RANDOM\_PORT or WebEnvironment.DEFINED\_PORT, you can just inject a fully configured TestRestTemplate and start using it. If necessary, additional customizations can be applied via the RestTemplateBuilder bean. Any URLs that do not specify a host and port will automatically connect to the embedded server:

@RunWith(SpringRunner.class) @SpringBootTest

**public class** MyTest {

@Autowired

**private** TestRestTemplate template;

@Test

**public void** testRequest() **throws** Exception {

HttpHeaders headers = template.getForEntity(***"/example"***, String.**class**).getHeaders(); assertThat(headers.getLocation().toString(), containsString(***"myotherhost"***));

}

@TestConfiguration

**static class** Config {

@Bean

**public** RestTemplateBuilder restTemplateBuilder() {

**return new** RestTemplateBuilder()

.additionalMessageConverters(...)

.customizers(...);

}

}

}

# WebSockets

Spring Boot provides WebSockets auto-configuration for embedded Tomcat (8 and 7), Jetty 9 and Undertow. If you’re deploying a war file to a standalone container, Spring Boot assumes that the container will be responsible for the configuration of its WebSocket support.

Spring Framework provides [rich WebSocket support](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#websocket) that can be easily accessed via the spring- boot-starter-websocket module.

# Web Services

Spring Boot provides Web Services auto-configuration so that all is required is defining your

Endpoints.

The [Spring Web Services features](http://docs.spring.io/spring-ws/docs/2.4.0.RELEASE/reference/htmlsingle) can be easily accessed via the spring-boot-starter- webservices module.

# Creating your own auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked-up by Spring Boot.

Auto-configuration can be associated to a "starter" that provides the auto-configuration code as well as the typical libraries that you would use with it. We will first cover what you need to know to build your own auto-configuration and we will move on to the [typical steps required to create a custom starter](#_bookmark292).

**Tip**

A [demo project](https://github.com/snicoll-demos/spring-boot-master-auto-configuration) is available to showcase how you can create a starter step by step.

## Understanding auto-configured beans

Under the hood, auto-configuration is implemented with standard @Configuration classes. Additional @Conditional annotations are used to constrain when the auto-configuration should apply. Usually auto-configuration classes use @ConditionalOnClass and @ConditionalOnMissingBean annotations. This ensures that auto-configuration only applies when relevant classes are found and when you have not declared your own @Configuration.

You can browse the source code of [spring-boot-autoconfigure](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure) to see the @Configuration

classes that we provide (see the [META-INF/spring.factories](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/resources/META-INF/spring.factories) file).

## Locating auto-configuration candidates

Spring Boot checks for the presence of a META-INF/spring.factories file within your published jar. The file should list your configuration classes under the EnableAutoConfiguration key.

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\ com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration

You can use the [@AutoConfigureAfter](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/AutoConfigureAfter.java) or [@AutoConfigureBefore](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/AutoConfigureBefore.java) annotations if your configuration needs to be applied in a specific order. For example, if you provide web-specific configuration, your class may need to be applied after WebMvcAutoConfiguration.

If you want to order certain auto-configurations that shouldn’t have any direct knowledge of each other, you can also use @AutoconfigureOrder. That annotation has the same semantic as the regular @Order annotation but provides a dedicated order for auto-configuration classes.

**Note**

Auto-configurations have to be loaded that way *only*. Make sure that they are defined in a specific package space and that they are never the target of component scan in particular.

## Condition annotations

You almost always want to include one or more @Conditional annotations on your auto-configuration class. The @ConditionalOnMissingBean is one common example that is used to allow developers to ‘override’ auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of @Conditional annotations that you can reuse in your own code by annotating @Configuration classes or individual @Bean methods.

### Class conditions

The @ConditionalOnClass and @ConditionalOnMissingClass annotations allows configuration to be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed using [ASM](http://asm.ow2.org/) you can actually use the value attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the name attribute if you prefer to specify the class name using a String value.

**Tip**

If you are using @ConditionalOnClass or @ConditionalOnMissingClass as a part of a meta-annotation to compose your own composed annotations you must use name as referring to the class in such a case is not handled.

### Bean conditions

The @ConditionalOnBean and @ConditionalOnMissingBean annotations allow a bean to be included based on the presence or absence of specific beans. You can use the value attribute to specify beans by type, or name to specify beans by name. The search attribute allows you to limit the ApplicationContext hierarchy that should be considered when searching for beans.

When placed on a @Bean method, the target type defaults to the return type of the method, for instance:

@Configuration

**public class** MyAutoConfiguration {

@Bean @ConditionalOnMissingBean

**public** MyService myService() { ... }

}

In the example above, the myService bean is going to be created if no bean of type MyService is already contained in the ApplicationContext.

**Tip**

You need to be very careful about the order that bean definitions are added as these conditions are evaluated based on what has been processed so far. For this reason, we recommend only using @ConditionalOnBean and @ConditionalOnMissingBean annotations on auto- configuration classes (since these are guaranteed to load after any user-defined beans definitions have been added).

**Note**

@ConditionalOnBean and @ConditionalOnMissingBean do not prevent @Configuration classes from being created. Using these conditions at the class level is equivalent to marking each contained @Bean method with the annotation.

### Property conditions

The @ConditionalOnProperty annotation allows configuration to be included based on a Spring Environment property. Use the prefix and name attributes to specify the property that should be checked. By default any property that exists and is not equal to false will be matched. You can also create more advanced checks using the havingValue and matchIfMissing attributes.

### Resource conditions

The @ConditionalOnResource annotation allows configuration to be included only when a specific resource is present. Resources can be specified using the usual Spring conventions, for example, file:/home/user/test.dat.

### Web application conditions

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations allow configuration to be included depending on whether the application is a 'web application'. A web application is any application that is using a Spring WebApplicationContext, defines a session scope or has a StandardServletEnvironment.

### SpEL expression conditions

The @ConditionalOnExpression annotation allows configuration to be included based on the result of a [SpEL expression](http://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#expressions).

## Creating your own starter

A full Spring Boot starter for a library may contain the following components:

* The autoconfigure module that contains the auto-configuration code.
* The starter module that provides a dependency to the autoconfigure module as well as the library and any additional dependencies that are typically useful. In a nutshell, adding the starter should be enough to start using that library.

**Tip**

You may combine the auto-configuration code and the dependency management in a single module if you don’t need to separate those two concerns.

### Naming

Please make sure to provide a proper namespace for your starter. Do not start your module names with spring-boot, even if you are using a different Maven groupId. We may offer an official support for the thing you’re auto-configuring in the future.

Here is a rule of thumb. Let’s assume that you are creating a starter for "acme", name the auto-configure module acme-spring-boot-autoconfigure and the starter acme-spring-boot-starter. If you only have one module combining the two, use acme-spring-boot-starter.

Besides, if your starter provides configuration keys, use a proper namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (e.g. server, management, spring,

etc). These are "ours" and we may improve/modify them in the future in such a way it could break your things.

Make sure to [trigger meta-data generation](#_bookmark586) so that IDE assistance is available for your keys as well. You may want to review the generated meta-data (META-INF/spring-configuration- metadata.json) to make sure your keys are properly documented.

### Autoconfigure module

The autoconfigure module contains everything that is necessary to get started with the library. It may also contain configuration keys definition (@ConfigurationProperties) and any callback interface that can be used to further customize how the components are initialized.

**Tip**

You should mark the dependencies to the library as optional so that you can include the autoconfigure module in your projects more easily. If you do it that way, the library won’t be provided and Spring Boot will back off by default.

### Starter module

The starter is an empty jar, really. Its only purpose is to provide the necessary dependencies to work with the library; see it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto- configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high as you should avoid bringing unnecessary dependencies for a typical usage of the library.

# What to read next

If you want to learn more about any of the classes discussed in this section you can check out the [Spring](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api) [Boot API documentation](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api) or you can browse the [source code directly](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE). If you have specific questions, take a look at the [how-to](#_bookmark438) section.

If you are comfortable with Spring Boot’s core features, you can carry on and read about [production-](#_bookmark297) [ready features](#_bookmark297).

**Part V. Spring Boot Actuator: Production-ready features**

Spring Boot includes a number of additional features to help you monitor and manage your application when it’s pushed to production. You can choose to manage and monitor your application using HTTP endpoints, with JMX or even by remote shell (SSH or Telnet). Auditing, health and metrics gathering can be automatically applied to your application.

Actuator HTTP endpoints are only available with a Spring MVC-based application. In particular, it will not work with Jersey [unless you enable Spring MVC as well.](#_bookmark532)

# Enabling production-ready features

The [spring-boot-actuator](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator) module provides all of Spring Boot’s production-ready features. The simplest way to enable the features is to add a dependency to the spring-boot-starter-actuator ‘Starter’.

|  |
| --- |
| **Definition of Actuator**  An actuator is a manufacturing term, referring to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change. |

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

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**</dependencies>**

For Gradle, use the declaration:

dependencies {

compile(***"org.springframework.boot:spring-boot-starter-actuator"***)

}

# Endpoints

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

The way that endpoints are exposed will depend on the type of technology that you choose. Most applications choose HTTP monitoring, where the ID of the endpoint is mapped to a URL. For example, by default, the health endpoint will be mapped to /health.

The following technology agnostic endpoints are available:

|  |  |  |
| --- | --- | --- |
| **ID** | **Description** | **Sensitive Default** |
| actuator  auditevents | Provides a hypermedia-based “discovery page” for the other endpoints. Requires Spring HATEOAS to be on the classpath.  Exposes audit events information for the current application. | true  true |
| autoconfig | Displays an auto-configuration report showing all auto- configuration candidates and the reason why they ‘were’ or ‘were not’ applied. | true |
| beans | Displays a complete list of all the Spring beans in your application. | true |
| configprops | Displays a collated list of all @ConfigurationProperties. | true |
| dump env | Performs a thread dump.  Exposes properties from Spring’s  ConfigurableEnvironment. | true true |
| flyway | Shows any Flyway database migrations that have been applied. | true |
| health | Shows application health information (when the application is secure, a simple ‘status’ when accessed over an unauthenticated connection or full message details when authenticated). | false |
| info loggers | Displays arbitrary application info.  Shows and modifies the configuration of loggers in the application. | false true |
| liquibase | Shows any Liquibase database migrations that have been applied. | true |
| metrics | Shows ‘metrics’ information for the current application. | true |
| mappings | Displays a collated list of all @RequestMapping paths. | true |

|  |  |  |
| --- | --- | --- |
| **ID** | **Description** | **Sensitive Default** |
| shutdown | Allows the application to be gracefully shutdown (not enabled | true |
|  | by default). |  |
| trace | Displays trace information (by default the last 100 HTTP | true |
|  | requests). |  |

If you are using Spring MVC, the following additional endpoints can also be used:

|  |  |  |
| --- | --- | --- |
| **ID**  docs | **Description**  Displays documentation, including example requests and responses, for the Actuator’s endpoints. Requires spring- boot-actuator-docs to be on the classpath. | **Sensitive Default**  false |
| heapdump | Returns a GZip compressed hprof heap dump file. | true |
| jolokia | Exposes JMX beans over HTTP (when Jolokia is on the classpath). | true |
| logfile | Returns the contents of the logfile (if logging.file or logging.path properties have been set). Supports the use of the HTTP Range header to retrieve part of the log file’s content. | true |

**Note**

Depending on how an endpoint is exposed, the sensitive property may be used as a security hint. For example, sensitive endpoints will require a username/password when they are accessed over HTTP (or simply disabled if web security is not enabled).

## Customizing endpoints

Endpoints can be customized using Spring properties. You can change if an endpoint is enabled, if it is considered sensitive and even its id.

For example, here is an application.properties that changes the sensitivity and id of the beans

endpoint and also enables shutdown.

**endpoints.beans.id**=springbeans **endpoints.beans.sensitive**=false **endpoints.shutdown.enabled**=true

**Note**

The prefix #endpoints + . + name” is used to uniquely identify the endpoint that is being configured.

By default, all endpoints except for shutdown are enabled. If you prefer to specifically “opt-in” endpoint enablement you can use the endpoints.enabled property. For example, the following will disable *all* endpoints except for info:

**endpoints.enabled**=false **endpoints.info.enabled**=true

Likewise, you can also choose to globally set the “sensitive” flag of all endpoints. By default, the sensitive flag depends on the type of endpoint (see the table above). For example, to mark *all* endpoints as sensitive except info:

**endpoints.sensitive**=true **endpoints.info.sensitive**=false

## Hypermedia for actuator MVC endpoints

If endpoints.hypermedia.enabled is set to true and [Spring HATEOAS](http://projects.spring.io/spring-hateoas) is on the classpath (e.g. through the spring-boot-starter-hateoas or if you are using [Spring Data REST](http://projects.spring.io/spring-data-rest)) then the HTTP endpoints from the Actuator are enhanced with hypermedia links, and a “discovery page” is added with links to all the endpoints. The “discovery page” is available on /actuator by default. It is implemented as an endpoint, allowing properties to be used to configure its path (endpoints.actuator.path) and whether or not it is enabled (endpoints.actuator.enabled).

When a custom management context path is configured, the “discovery page” will automatically move from /actuator to the root of the management context. For example, if the management context path is /management then the discovery page will be available from /management.

If the [HAL Browser](https://github.com/mikekelly/hal-browser) is on the classpath via its webjar (org.webjars:hal-browser), or via the spring-data-rest-hal-browser then an HTML “discovery page”, in the form of the HAL Browser, is also provided.

## CORS support

[Cross-origin resource sharing](http://en.wikipedia.org/wiki/Cross-origin_resource_sharing) (CORS) is a [W3C specification](http://www.w3.org/TR/cors/) that allows you to specify in a flexible way what kind of cross domain requests are authorized. Actuator’s MVC endpoints can be configured to support such scenarios.

CORS support is disabled by default and is only enabled once the endpoints.cors.allowed- origins property has been set. The configuration below permits GET and POST calls from the example.com domain:

**endpoints.cors.allowed-origins**[=http://example.com](http://example.com/) **endpoints.cors.allowed-methods**=GET,POST

**Tip**

Check [EndpointCorsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointCorsProperties.java) for a complete list of options.

## Adding custom endpoints

If you add a @Bean of type Endpoint then it will automatically be exposed over JMX and HTTP (if there is an server available). An HTTP endpoints can be customized further by creating a bean of type MvcEndpoint. Your MvcEndpoint is not a @Controller but it can use @RequestMapping (and @Managed\*) to expose resources.

**Tip**

If you are doing this as a library feature consider adding a configuration class annotated with @ManagementContextConfiguration to /META-INF/spring.factories under the key org.springframework.boot.actuate.autoconfigure.ManagementContextConfiguration.

If you do that then the endpoint will move to a child context with all the other MVC endpoints if your users ask for a separate management port or address. A configuration declared this way can be a WebConfigurerAdapter if it wants to add static resources (for instance) to the management endpoints.

## Health information

Health information can be used to check the status of your running application. It is often used by monitoring software to alert someone if a production system goes down. The default information exposed by the health endpoint depends on how it is accessed. For an unauthenticated connection in a secure application a simple ‘status’ message is returned, and for an authenticated connection additional details are also displayed (see [Section 48.7, “HTTP health endpoint format and access restrictions”](#_bookmark321) for HTTP details).

Health information is collected from all [HealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthIndicator.java) beans defined in your ApplicationContext. Spring Boot includes a number of auto-configured HealthIndicators and you can also write your own. By default, the final system state is derived by the HealthAggregator which sorts the statuses from each HealthIndicator based on an ordered list of statuses. The first status in the sorted list is used as the overall health status. If no HealthIndicator returns a status that is known to the HealthAggregator, an UNKNOWN status is used.

## Security with HealthIndicators

Information returned by HealthIndicators is often somewhat sensitive in nature. For example, you probably don’t want to publish details of your database server to the world. For this reason, by default, only the health status is exposed over an unauthenticated HTTP connection. If you are happy for complete health information to always be exposed you can set endpoints.health.sensitive to false.

Health responses are also cached to prevent “denial of service” attacks. Use the endpoints.health.time-to-live property if you want to change the default cache period of 1000 milliseconds.

### Auto-configured HealthIndicators

The following HealthIndicators are auto-configured by Spring Boot when appropriate:

|  |
| --- |
| **Name Description** |
| [CassandraHealthICnhdeickcsatthoart](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/CassandraHealthIndicator.java) a Cassandra database is up. |
| [DiskSpaceHealthICnhdeickcsatfoorrl](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/DiskSpaceHealthIndicator.java)ow disk space.  [DataSourceHealthCIhnedckiscathtaot ra](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/DataSourceHealthIndicator.java) connection to DataSource can be obtained. |
| [ElasticsearchHeaClhtehckIsndthiact aatnoEr](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/ElasticsearchHealthIndicator.java)lasticsearch cluster is up. |
| [JmsHealthIndicatCohre](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/JmsHealthIndicator.java)cks that a JMS broker is up. |

|  |
| --- |
| **Name Description** |
| [MailHealthIndicaCthoerc](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/MailHealthIndicator.java)ks that a mail server is up.  [MongoHealthIndicCahteockr](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/MongoHealthIndicator.java)s that a Mongo database is up. |
| [RabbitHealthIndiCchaetckosr](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/RabbitHealthIndicator.java)that a Rabbit server is up.  [RedisHealthIndicCahteockr](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/RedisHealthIndicator.java)s that a Redis server is up. |
| [SolrHealthIndicaCthoerc](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/SolrHealthIndicator.java)ks that a Solr server is up. |

**Tip**

It is possible to disable them all using the management.health.defaults.enabled property.

### Writing custom HealthIndicators

To provide custom health information you can register Spring beans that implement the [HealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthIndicator.java) interface. You need to provide an implementation of the health() method and return a Health response. The Health response should include a status and can optionally include additional details to be displayed.

**import** org.springframework.boot.actuate.health.Health;

**import** org.springframework.boot.actuate.health.HealthIndicator;

**import** org.springframework.stereotype.Component;

@Component

**public class** MyHealthIndicator **implements** HealthIndicator {

@Override

**public** Health health() {

**int** errorCode = check(); *// perform some specific health check*

**if** (errorCode != 0) {

**return** Health.down().withDetail(***"Error Code"***, errorCode).build();

}

**return** Health.up().build();

}

}

**Note**

The identifier for a given HealthIndicator is the name of the bean without the HealthIndicator suffix if it exists. In the example above, the health information will be available in an entry named my.

In addition to Spring Boot’s predefined [Status](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/Status.java) types, it is also possible for Health to return a custom Status that represents a new system state. In such cases a custom implementation of the [HealthAggregator](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthAggregator.java) interface also needs to be provided, or the default implementation has to be configured using the management.health.status.order configuration property.

For example, assuming a new Status with code FATAL is being used in one of your HealthIndicator implementations. To configure the severity order add the following to your application properties:

**management.health.status.order**=FATAL, DOWN, OUT\_OF\_SERVICE, UNKNOWN, UP

The HTTP status code in the response reflects the overall health status (e.g. UP maps to 200, OUT\_OF\_SERVICE or DOWN to 503). You might also want to register custom status mappings with the HealthMvcEndpoint if you access the health endpoint over HTTP. For example, the following maps FATAL to 503 (service unavailable):

**endpoints.health.mapping.FATAL**=503

The default status mappings for the built-in statuses are:

|  |  |
| --- | --- |
| **Status**  DOWN | **Mapping**  SERVICE\_UNAVAILABLE (503) |
| OUT\_OF\_SERVICE | SERVICE\_UNAVAILABLE (503) |
| UP  UNKNOWN | No mapping by default, so http status is 200  No mapping by default, so http status is 200 |

## Application information

Application information exposes various information collected from all [InfoContributor](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/InfoContributor.java) beans defined in your ApplicationContext. Spring Boot includes a number of auto-configured InfoContributors and you can also write your own.

### Auto-configured InfoContributors

The following InfoContributors are auto-configured by Spring Boot when appropriate:

|  |
| --- |
| **Name Description** |
| [EnvironmentInfoCEoxpnotsreibanuytokre](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/EnvironmentInfoContributor.java)y from the Environment under the info key. |
| [GitInfoContributEoxpr](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/GitInfoContributor.java)ose git information if a git.properties file is available. |
| [BuildInfoContribEuxptoosr](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/BuildInfoContributor.java)e build information if a META-INF/build-info.properties file is available. |

**Tip**

It is possible to disable them all using the management.info.defaults.enabled property.

### Custom application info information

You can customize the data exposed by the info endpoint by setting info.\* Spring properties. All Environment properties under the info key will be automatically exposed. For example, you could add the following to your application.properties:

**info.app.encoding**=UTF-8 **info.app.java.source**=1.8 **info.app.java.target**=1.8

**Tip**

Rather than hardcoding those values you could also [expand info properties at build time](#_bookmark446).

Assuming you are using Maven, you could rewrite the example above as follows:

[**info.app.encoding**=@project.build.sourceEncoding@](mailto:info.app.encoding%3D@project.build.sourceEncoding@) [**info.app.java.source**=@java.version@](mailto:info.app.java.source%3D@java.version@) [**info.app.java.target**=@java.version@](mailto:info.app.java.target%3D@java.version@)

### Git commit information

Another useful feature of the info endpoint is its ability to publish information about the state of your git source code repository when the project was built. If a GitProperties bean is available, the git.branch, git.commit.id and git.commit.time properties will be exposed.

**Tip**

A GitProperties bean is auto-configured if a git.properties file is available at the root of the classpath. See [Generate git information](#_bookmark549) for more details.

If you want to display the full git information (i.e. the full content of git.properties), use the

management.info.git.mode property:

**management.info.git.mode**=full

### Build information

The info endpoint can also publish information about your build if a BuildProperties bean is available. This happens if a META-INF/build-info.properties file is available in the classpath.

**Tip**

The Maven and Gradle plugins can both generate that file, see [Generate build information](#_bookmark548) for more details.

### Writing custom InfoContributors

To provide custom application information you can register Spring beans that implement the

[InfoContributor](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/InfoContributor.java) interface.

The example below contributes an example entry with a single value:

**import** java.util.Collections;

**import** org.springframework.boot.actuate.info.Info;

**import** org.springframework.boot.actuate.info.InfoContributor;

**import** org.springframework.stereotype.Component;

@Component

**public class** ExampleInfoContributor **implements** InfoContributor {

@Override

**public void** contribute(Info.Builder builder) { builder.withDetail(***"example"***,

Collections.singletonMap(***"key"***, ***"value"***));

}

}

If you hit the info endpoint you should see a response that contains the following additional entry:

**{**

***"example"***: **{**

***"key"*** : ***"value"***

**}**

**}**

# Monitoring and management over HTTP

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

## Accessing sensitive endpoints

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.

**Tip**

Use the management.security.roles property if you want something different to ACTUATOR.

If you are deploying applications behind a firewall, you may prefer that all your actuator endpoints can be accessed without requiring authentication. You can do this by changing the management.security.enabled property:

**application.properties.**

**management.security.enabled**=false

**Note**

By default, actuator endpoints are exposed on the same port that serves regular HTTP traffic. Take care not to accidentally expose sensitive information if you change the management.security.enabled property.

If you’re deploying applications publicly, you may want to add ‘Spring Security’ to handle user authentication. When ‘Spring Security’ is added, by default ‘basic’ authentication will be used with the username user and a generated password (which is printed on the console when the application starts).

**Tip**

Generated passwords are logged as the application starts. Search for ‘Using default security password’.

You can use Spring properties to change the username and password and to change the security role(s) required to access the endpoints. For example, you might set the following in your application.properties:

**security.user.name**=admin **security.user.password**=secret **management.security.roles**=SUPERUSER

If your application has custom security configuration and you want all your actuator endpoints to be accessible without authentication, you need to explicitly configure that in your security configuration. Along with that, you need to change the management.security.enabled property to false.

If your custom security configuration secures your actuator endpoints, you also need to ensure that the authenticated user has the roles specified under management.security.roles.

**Tip**

If you don’t have a use case for exposing basic health information to unauthenticated users, and you have secured the actuator endpoints with custom security, you can set management.security.enabled to false. This will inform Spring Boot to skip the additional role check.

## Customizing the management endpoint paths

Sometimes it is useful to group all management endpoints under a single path. For example, your application might already use /info for another purpose. You can use the management.context- path property to set a prefix for your management endpoint:

**management.context-path**=/manage

The application.properties example above will change the endpoint from /{id} to /manage/

{id} (e.g. /manage/info).

**Note**

Unless the management port has been configured to [expose endpoints using a different HTTP](#_bookmark317) [port](#_bookmark317), management.context-path is relative to server.context-path.

You can also change the “id” of an endpoint (using endpoints.{name}.id) which then changes the default resource path for the MVC endpoint. Legal endpoint ids are composed only of alphanumeric characters (because they can be exposed in a number of places, including JMX object names, where special characters are forbidden). The MVC path can be changed separately by configuring endpoints.{name}.path, and there is no validation on those values (so you can use anything that is legal in a URL path). For example, to change the location of the /health endpoint to /ping/me you can set endpoints.health.path=/ping/me.

**Note**

Even if an endpoint path is configured separately, it is still relative to the management.context- path.

**Tip**

If you provide a custom MvcEndpoint remember to include a settable path property, and default it to /{id} if you want your code to behave like the standard MVC endpoints. (Take a look at the HealthMvcEndpoint to see how you might do that.) If your custom endpoint is an Endpoint (not an MvcEndpoint) then Spring Boot will take care of the path for you.

## Customizing the management server port

Exposing management endpoints using the default HTTP port is a sensible choice for cloud based deployments. If, however, your application runs inside your own data center you may prefer to expose endpoints using a different HTTP port.

The management.port property can be used to change the HTTP port.

**management.port**=8081

Since your management port is often protected by a firewall, and not exposed to the public you might not need security on the management endpoints, even if your main application is secure. In that case you will have Spring Security on the classpath, and you can disable management security like this:

**management.security.enabled**=false

(If you don’t have Spring Security on the classpath then there is no need to explicitly disable the management security in this way, and it might even break the application.)

## Configuring management-specific SSL

When configured to use a custom port, the management server can also be configured with its own SSL using the various management.ssl.\* properties. For example, this allows a management server to be available via HTTP while the main application uses HTTPS:

**server.port**=8443 **server.ssl.enabled**=true

**server.ssl.key-store**=classpath:store.jks **server.ssl.key-password**=secret **management.port**=8080 **management.ssl.enabled**=false

Alternatively, both the main server and the management server can use SSL but with different key stores:

**server.port**=8443 **server.ssl.enabled**=true

**server.ssl.key-store**=classpath:main.jks **server.ssl.key-password**=secret **management.port**=8080 **management.ssl.enabled**=true

**management.ssl.key-store**=classpath:management.jks **management.ssl.key-password**=secret

## Customizing the management server address

You can customize the address that the management endpoints are available on by setting the management.address property. This can be useful if you want to listen only on an internal or ops- facing network, or to only listen for connections from localhost.

**Note**

You can only listen on a different address if the port is different to the main server port.

Here is an example application.properties that will not allow remote management connections:

**management.port**=8081 **management.address**=127.0.0.1

## Disabling HTTP endpoints

If you don’t want to expose endpoints over HTTP you can set the management port to -1:

**management.port**=-1

## HTTP health endpoint format and access restrictions

The information exposed by the health endpoint varies depending on whether or not it’s accessed anonymously, and whether or not the enclosing application is secure. By default, when accessed anonymously in a secure application, any details about the server’s health are hidden and the endpoint will simply indicate whether or not the server is up or down. Furthermore the response is cached for a configurable period to prevent the endpoint being used in a denial of service attack. The endpoints.health.time-to-live property is used to configure the caching period in milliseconds. It defaults to 1000, i.e. one second.

Sample summarized HTTP response (default for anonymous request):

$ curl -i localhost:8080/health HTTP/1.1 200

X-Application-Context: application

Content-Type: application/vnd.spring-boot.actuator.v1+json;charset=UTF-8 Content-Length: 15

{"status":"UP"}

Sample summarized HTTP response for status "DOWN" (notice the 503 status code):

$ curl -i localhost:8080/health HTTP/1.1 503

X-Application-Context: application

Content-Type: application/vnd.spring-boot.actuator.v1+json;charset=UTF-8 Content-Length: 17

{"status":"DOWN"}

Sample detailed HTTP response:

$ curl -i localhost:8080/health HTTP/1.1 200 OK

X-Application-Context: application

Content-Type: application/vnd.spring-boot.actuator.v1+json;charset=UTF-8 Content-Length: 221

{

"status" : "UP", "diskSpace" : {

"status" : "UP", "total" : 63251804160,

"free" : 31316164608,

"threshold" : 10485760

},

"db" : {

"status" : "UP",

"database" : "H2", "hello" : 1

}

}

The above-described restrictions can be enhanced, thereby allowing only authenticated users full access to the health endpoint in a secure application. To do so, set endpoints.health.sensitive to true. Here’s a summary of behavior (with default sensitive flag value “false” indicated in bold):

|  |  |  |
| --- | --- | --- |
| **management.securit** | **ye.nednpaobilnetds.health.seUnsniauttihveenticated** | **Authenticated (with right role)** |
| false | \* Full content | Full content |

|  |  |  |
| --- | --- | --- |
| **management.securit** | **ye.nednpaobilnetds.health.seUnsniauttihveenticated** | **Authenticated (with right role)** |
| true  true | **false** Status only  true No content | Full content  Full content |

# Monitoring and management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default Spring Boot will expose management endpoints as JMX MBeans under the org.springframework.boot domain.

## Customizing MBean names

The name of the MBean is usually generated from the id of the endpoint. For example the health

endpoint is exposed as org.springframework.boot/Endpoint/healthEndpoint.

If your application contains more than one Spring ApplicationContext you may find that names clash. To solve this problem you can set the endpoints.jmx.unique-names property to true so that MBean names are always unique.

You can also customize the JMX domain under which endpoints are exposed. Here is an example

application.properties:

**endpoints.jmx.domain**=myapp **endpoints.jmx.unique-names**=true

## Disabling JMX endpoints

If you don’t want to expose endpoints over JMX you can set the endpoints.jmx.enabled property to false:

**endpoints.jmx.enabled**=false

## Using Jolokia for JMX over HTTP

Jolokia is a JMX-HTTP bridge giving an alternative method of accessing JMX beans. To use Jolokia, simply include a dependency to org.jolokia:jolokia-core. For example, using Maven you would add the following:

**<dependency>**

**<groupId>**org.jolokia**</groupId>**

**<artifactId>**jolokia-core**</artifactId>**

**</dependency>**

Jolokia can then be accessed using /jolokia on your management HTTP server.

### Customizing Jolokia

Jolokia has a number of settings that you would traditionally configure using servlet parameters. With Spring Boot you can use your application.properties, simply prefix the parameter with jolokia.config.:

**jolokia.config.debug**=true

### Disabling Jolokia

If you are using Jolokia but you don’t want Spring Boot to configure it, simply set the

endpoints.jolokia.enabled property to false:

**endpoints.jolokia.enabled**=false

# Monitoring and management using a remote shell (deprecated)

Spring Boot supports an integrated Java shell called ‘CRaSH’. You can use CRaSH to ssh or telnet into your running application. To enable remote shell support, add the following dependency to your project:

**<dependency>**

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

**<artifactId>**spring-boot-starter-remote-shell**</artifactId>**

**</dependency>**

**Note**

The remote shell is deprecated and will be removed in Spring Boot 2.0.

**Tip**

If you want to also enable telnet access you will additionally need a dependency on

org.crsh:crsh.shell.telnet.

**Note**

CRaSH requires to run with a JDK as it compiles commands on the fly. If a basic help command fails, you are probably running with a JRE.

## Connecting to the remote shell

By default the remote shell will listen for connections on port 2000. The default user is user and the default password will be randomly generated and displayed in the log output. If your application is using Spring Security, the shell will use [the same configuration](#_bookmark156) by default. If not, a simple authentication will be applied and you should see a message like this:

Using default password for shell access: ec03e16c-4cf4-49ee-b745-7c8255c1dd7e

Linux and OSX users can use ssh to connect to the remote shell, Windows users can download and install [PuTTY](http://www.putty.org/).

/\\ / '\_ \_ \_(\_)\_ \_ \ \ \ \ ( ( )\ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ )| |\_)| | | | | || (\_| | ) ) ) ) ' | | . |\_| |\_|\_| |\_\ , | / / / /

=========|\_|==============| /=/\_/\_/\_/

:: Spring Boot :: (v1.5.8.RELEASE) on myhost

\_ \_

$ ssh -p 2000 user@localhost

user@localhost's password:

. \_

Type help for a list of commands. Spring Boot provides metrics, beans, autoconfig and endpoint

commands.

### Remote shell credentials

You can use the management.shell.auth.simple.user.name and management.shell.auth.simple.user.password properties to configure custom connection credentials. It is also possible to use a ‘Spring Security’ AuthenticationManager to handle login duties. See the [CrshAutoConfiguration](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/CrshAutoConfiguration.html) and [ShellProperties](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/ShellProperties.html) Javadoc for full details.

## Extending the remote shell

The remote shell can be extended in a number of interesting ways.

### Remote shell commands

You can write additional shell commands using Groovy (see the CRaSH documentation for details). Due to limitations in CRaSH’s Java compiler, commands written in Java are not supported. By default Spring Boot will search for commands in the following locations:

* classpath\*:/commands/\*\*
* classpath\*:/crash/commands/\*\*

**Tip**

You can change the search path by settings a shell.command-path-patterns property.

**Note**

If you are using an executable archive, any classes that a shell command depends upon must be packaged in a nested jar rather than directly in the executable jar or war.

Here is a simple ‘hello’ command that could be loaded from src/main/resources/commands/ hello.groovy

**package** commands

**import** org.crsh.cli.Command

**import** org.crsh.cli.Usage

**import** org.crsh.command.InvocationContext

**class** hello {

@Usage("Say Hello") @Command

def main(InvocationContext context) {

**return *"Hello"***

}

}

Spring Boot adds some additional attributes to InvocationContext that you can access from your command:

|  |
| --- |
| **Attribute Name Description** |
| spring.boot.version The version of Spring Boot  spring.version The version of the core Spring Framework |

|  |
| --- |
| **Attribute Name Description** |
| spring.beanfactory Access to the Spring BeanFactory  spring.environment Access to the Spring Environment |

### Remote shell plugins

In addition to new commands, it is also possible to extend other CRaSH shell features. All Spring Beans that extend org.crsh.plugin.CRaSHPlugin will be automatically registered with the shell.

For more information please refer to the [CRaSH reference documentation](http://www.crashub.org/).

# Loggers

Spring Boot Actuator includes the ability to view and configure the log levels of your application at runtime. You can view either the entire list or an individual logger’s configuration which is made up of both the explicitly configured logging level as well as the effective logging level given to it by the logging framework. These levels can be:

* TRACE
* DEBUG
* INFO
* WARN
* ERROR
* FATAL
* OFF
* null

with null indicating that there is no explicit configuration.

## Configure a Logger

In order to configure a given logger, you POST a partial entity to the resource’s URI:

**{**

***"configuredLevel"***: ***"DEBUG"***

**}**

# Metrics

Spring Boot Actuator includes a metrics service with ‘gauge’ and ‘counter’ support. A ‘gauge’ records a single value; and a ‘counter’ records a delta (an increment or decrement). Spring Boot Actuator also provides a [PublicMetrics](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/endpoint/PublicMetrics.java) interface that you can implement to expose metrics that you cannot record via one of those two mechanisms. Look at [SystemPublicMetrics](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/endpoint/SystemPublicMetrics.java) for an example.

Metrics for all HTTP requests are automatically recorded, so if you hit the metrics endpoint you should see a response similar to this:

**{**

***"counter.status.200.root"***: 20**,**

***"counter.status.200.metrics"***: 3**,**

***"counter.status.200.star-star"***: 5**,**

***"counter.status.401.root"***: 4**,**

***"gauge.response.star-star"***: 6**,**

***"gauge.response.root"***: 2**,**

***"gauge.response.metrics"***: 3**,**

***"classes"***: 5808**,**

***"classes.loaded"***: 5808**,**

***"classes.unloaded"***: 0**,**

***"heap"***: 3728384**,**

***"heap.committed"***: 986624**,**

***"heap.init"***: 262144**,**

***"heap.used"***: 52765**,**

***"nonheap"***: 0**,**

***"nonheap.committed"***: 77568**,**

***"nonheap.init"***: 2496**,**

***"nonheap.used"***: 75826**,**

***"mem"***: 986624**,**

***"mem.free"***: 933858**,**

***"processors"***: 8**,**

***"threads"***: 15**,**

***"threads.daemon"***: 11**,**

***"threads.peak"***: 15**,**

***"threads.totalStarted"***: 42**,**

***"uptime"***: 494836**,**

***"instance.uptime"***: 489782**,**

***"datasource.primary.active"***: 5**,**

***"datasource.primary.usage"***: 0.25

**}**

Here we can see basic memory, heap, class loading, processor and thread pool information along with some HTTP metrics. In this instance the root (‘/’) and /metrics URLs have returned HTTP 200 responses 20 and 3 times respectively. It also appears that the root URL returned HTTP 401 (unauthorized) 4 times. The double asterisks (star-star) comes from a request matched by Spring MVC as /\*\* (normally a static resource).

The gauge shows the last response time for a request. So the last request to root took 2ms to respond and the last to /metrics took 3ms.

**Note**

In this example we are actually accessing the endpoint over HTTP using the /metrics URL, this explains why metrics appears in the response.

## System metrics

The following system metrics are exposed by Spring Boot:

* The total system memory in KB (mem)
* The amount of free memory in KB (mem.free)
* The number of processors (processors)
* The system uptime in milliseconds (uptime)
* The application context uptime in milliseconds (instance.uptime)
* The average system load (systemload.average)
* Heap information in KB (heap, heap.committed, heap.init, heap.used)
* Thread information (threads, thread.peak, thread.daemon)
* Class load information (classes, classes.loaded, classes.unloaded)
* Garbage collection information (gc.xxx.count, gc.xxx.time)

## DataSource metrics

The following metrics are exposed for each supported DataSource defined in your application:

* The number of active connections (datasource.xxx.active)
* The current usage of the connection pool (datasource.xxx.usage).

All data source metrics share the datasource. prefix. The prefix is further qualified for each data source:

* If the data source is the primary data source (that is either the only available data source or the one flagged @Primary amongst the existing ones), the prefix is datasource.primary.
* If the data source bean name ends with DataSource, the prefix is the name of the bean without

DataSource (i.e. datasource.batch for batchDataSource).

* In all other cases, the name of the bean is used.

It is possible to override part or all of those defaults by registering a bean with a customized version of DataSourcePublicMetrics. By default, Spring Boot provides metadata for all supported data sources; you can add additional DataSourcePoolMetadataProvider beans if your favorite data source isn’t supported out of the box. See DataSourcePoolMetadataProvidersConfiguration for examples.

## Cache metrics

The following metrics are exposed for each supported cache defined in your application:

* The current size of the cache (cache.xxx.size)
* Hit ratio (cache.xxx.hit.ratio)
* Miss ratio (cache.xxx.miss.ratio)

**Note**

Cache providers do not expose the hit/miss ratio in a consistent way. While some expose an

**aggregated** value (i.e. the hit ratio since the last time the stats were cleared), others expose a

**temporal** value (i.e. the hit ratio of the last second). Check your caching provider documentation for more details.

If two different cache managers happen to define the same cache, the name of the cache is prefixed by the name of the CacheManager bean.

It is possible to override part or all of those defaults by registering a bean with a customized version of CachePublicMetrics. By default, Spring Boot provides cache statistics for EhCache, Hazelcast, Infinispan, JCache and Guava. You can add additional CacheStatisticsProvider beans if your favorite caching library isn’t supported out of the box. See CacheStatisticsAutoConfiguration for examples.

## Tomcat session metrics

If you are using Tomcat as your embedded servlet container, session metrics will automatically be exposed. The httpsessions.active and httpsessions.max keys provide the number of active and maximum sessions.

## Recording your own metrics

To record your own metrics inject a [CounterService](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/CounterService.java) and/or [GaugeService](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/GaugeService.java) into your bean. The CounterService exposes increment, decrement and reset methods; the GaugeService provides a submit method.

Here is a simple example that counts the number of times that a method is invoked:

**import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.boot.actuate.metrics.CounterService; **import** org.springframework.stereotype.Service;

@Service

**public class** MyService {

**private final** CounterService counterService; @Autowired

**public** MyService(CounterService counterService) {

**this**.counterService = counterService;

}

**public void** exampleMethod() {

**this**.counterService.increment(***"services.system.myservice.invoked"***);

}

}

**Tip**

You can use any string as a metric name but you should follow guidelines of your chosen store/ graphing technology. Some good guidelines for Graphite are available on [Matt Aimonetti’s Blog](http://matt.aimonetti.net/posts/2013/06/26/practical-guide-to-graphite-monitoring/).

## Adding your own public metrics

To add additional metrics that are computed every time the metrics endpoint is invoked, simply register additional PublicMetrics implementation bean(s). By default, all such beans are gathered by the endpoint. You can easily change that by defining your own MetricsEndpoint.

## Special features with Java 8

The default implementation of GaugeService and CounterService provided by Spring Boot depends on the version of Java that you are using. With Java 8 (or better) the implementation switches to a high-performance version optimized for fast writes, backed by atomic in-memory buffers, rather than by the immutable but relatively expensive Metric<?> type (counters are approximately 5 times faster and gauges approximately twice as fast as the repository-based implementations). The Dropwizard metrics services (see below) are also very efficient even for Java 7 (they have backports of some of the Java 8 concurrency libraries), but they do not record timestamps for metric values. If performance of metric gathering is a concern then it is always advisable to use one of the high-performance options, and also to only read metrics infrequently, so that the writes are buffered locally and only read when needed.

**Note**

The old MetricRepository and its InMemoryMetricRepository implementation are not used by default if you are on Java 8 or if you are using Dropwizard metrics.

## Metric writers, exporters and aggregation

Spring Boot provides a couple of implementations of a marker interface called Exporter which can be used to copy metric readings from the in-memory buffers to a place where they can be analyzed and displayed. Indeed, if you provide a @Bean that implements the MetricWriter interface (or GaugeWriter for simple use cases) and mark it @ExportMetricWriter, then it will automatically be hooked up to an Exporter and fed metric updates every 5 seconds (configured via spring.metrics.export.delay-millis). In addition, any MetricReader that you define and mark as @ExportMetricReader will have its values exported by the default exporter.

**Note**

This feature is enabling scheduling in your application (@EnableScheduling) which can be a problem if you run an integration test as your own scheduled tasks will start. You can disable this behaviour by setting spring.metrics.export.enabled to false.

The default exporter is a MetricCopyExporter which tries to optimize itself by not copying values that haven’t changed since it was last called (the optimization can be switched off using a flag spring.metrics.export.send-latest). Note also that the Dropwizard MetricRegistry has no support for timestamps, so the optimization is not available if you are using Dropwizard metrics (all metrics will be copied on every tick).

The default values for the export trigger (delay-millis, includes, excludes and send-latest) can be set as spring.metrics.export.\*. Individual values for specific MetricWriters can be set as spring.metrics.export.triggers.<name>.\* where <name> is a bean name (or pattern for matching bean names).

**Warning**

The automatic export of metrics is disabled if you switch off the default MetricRepository (e.g. by using Dropwizard metrics). You can get back the same functionality be declaring a bean of your own of type MetricReader and declaring it to be @ExportMetricReader.

### Example: Export to Redis

If you provide a @Bean of type RedisMetricRepository and mark it @ExportMetricWriter the metrics are exported to a Redis cache for aggregation. The RedisMetricRepository has two important parameters to configure it for this purpose: prefix and key (passed into its constructor). It is best to use a prefix that is unique to the application instance (e.g. using a random value and maybe the logical name of the application to make it possible to correlate with other instances of the same application). The “key” is used to keep a global index of all metric names, so it should be unique “globally”, whatever that means for your system (e.g. two instances of the same system could share a Redis cache if they have distinct keys).

Example:

@Bean @ExportMetricWriter

MetricWriter metricWriter(MetricExportProperties export) {

**return new** RedisMetricRepository(connectionFactory, export.getRedis().getPrefix(), export.getRedis().getKey());

}

**application.properties.**

**spring.metrics.export.redis.prefix**: metrics.mysystem.${spring.application.name:application}.

${random.value:0000}

**spring.metrics.export.redis.key**: keys.metrics.mysystem

The prefix is constructed with the application name and id at the end, so it can easily be used to identify a group of processes with the same logical name later.

**Note**

It’s important to set both the key and the prefix. The key is used for all repository operations, and can be shared by multiple repositories. If multiple repositories share a key (like in the case where you need to aggregate across them), then you normally have a read-only “master” repository that has a short, but identifiable, prefix (like “metrics.mysystem”), and many write-only repositories with prefixes that start with the master prefix (like metrics.mysystem.\* in the example above). It is efficient to read all the keys from a “master” repository like that, but inefficient to read a subset with a longer prefix (e.g. using one of the writing repositories).

**Tip**

The example above uses MetricExportProperties to inject and extract the key and prefix. This is provided to you as a convenience by Spring Boot, configured with sensible defaults. There is nothing to stop you using your own values as long as they follow the recommendations.

### Example: Export to Open TSDB

If you provide a @Bean of type OpenTsdbGaugeWriter and mark it @ExportMetricWriter metrics are exported to [Open TSDB](http://opentsdb.net/) for aggregation. The OpenTsdbGaugeWriter has a url property that you need to set to the Open TSDB “/put” endpoint, e.g. localhost:4242/api/put). It also has a namingStrategy that you can customize or configure to make the metrics match the data structure you need on the server. By default it just passes through the metric name as an Open TSDB metric name, and adds the tags “domain” (with value “org.springframework.metrics”) and “process” (with the

value equal to the object hash of the naming strategy). Thus, after running the application and generating some metrics you can inspect the metrics in the TSD UI (localhost:4242 by default).

Example:

curl localhost:4242/api/query?start=1h-ago&m=max:counter.status.200.root [

{

"metric": "counter.status.200.root", "tags": {

"domain": "org.springframework.metrics", "process": "b968a76"

},

"aggregateTags": [], "dps": {

"1430492872": 2,

"1430492875": 6

}

}

]

### Example: Export to Statsd

To export metrics to Statsd, make sure first that you have added com.timgroup:java-statsd- client as a dependency of your project (Spring Boot provides a dependency management for it). Then add a spring.metrics.export.statsd.host value to your application.properties file. Connections will be opened to port 8125 unless a spring.metrics.export.statsd.port override is provided. You can use spring.metrics.export.statsd.prefix if you want a custom prefix.

Alternatively, you can provide a @Bean of type StatsdMetricWriter and mark it

@ExportMetricWriter:

@Value("${spring.application.name:application}.${random.value:0000}")

**private** String prefix = ***"metrics"***;

@Bean @ExportMetricWriter

MetricWriter metricWriter() {

**return new** StatsdMetricWriter(prefix, ***"localhost"***, 8125);

}

### Example: Export to JMX

If you provide a @Bean of type JmxMetricWriter marked @ExportMetricWriter the metrics are exported as MBeans to the local server (the MBeanExporter is provided by Spring Boot JMX auto- configuration as long as it is switched on). Metrics can then be inspected, graphed, alerted etc. using any tool that understands JMX (e.g. JConsole or JVisualVM).

Example:

@Bean @ExportMetricWriter

MetricWriter metricWriter(MBeanExporter exporter) {

**return new** JmxMetricWriter(exporter);

}

Each metric is exported as an individual MBean. The format for the ObjectNames is given by an ObjectNamingStrategy which can be injected into the JmxMetricWriter (the default breaks up the metric name and tags the first two period-separated sections in a way that should make the metrics group nicely in JVisualVM or JConsole).

## Aggregating metrics from multiple sources

There is an AggregateMetricReader that you can use to consolidate metrics from different physical sources. Sources for the same logical metric just need to publish them with a period-separated prefix, and the reader will aggregate (by truncating the metric names, and dropping the prefix). Counters are summed and everything else (i.e. gauges) take their most recent value.

This is very useful if multiple application instances are feeding to a central (e.g. Redis) repository and you want to display the results. Particularly recommended in conjunction with a MetricReaderPublicMetrics for hooking up to the results to the “/metrics” endpoint.

Example:

@Autowired

**private** MetricExportProperties export;

@Bean

**public** PublicMetrics metricsAggregate() {

**return new** MetricReaderPublicMetrics(aggregatesMetricReader());

}

**private** MetricReader globalMetricsForAggregation() {

**return new** RedisMetricRepository(**this**.connectionFactory,

**this**.export.getRedis().getAggregatePrefix(), **this**.export.getRedis().getKey());

}

**private** MetricReader aggregatesMetricReader() { AggregateMetricReader repository = **new** AggregateMetricReader(

globalMetricsForAggregation());

**return** repository;

}

**Note**

The example above uses MetricExportProperties to inject and extract the key and prefix. This is provided to you as a convenience by Spring Boot, and the defaults will be sensible. They are set up in MetricExportAutoConfiguration.

**Note**

The MetricReaders above are not @Beans and are not marked as @ExportMetricReader because they are just collecting and analyzing data from other repositories, and don’t want to export their values.

## Dropwizard Metrics

A default MetricRegistry Spring bean will be created when you declare a dependency to the io.dropwizard.metrics:metrics-core library; you can also register you own @Bean instance if you need customizations. Users of the [Dropwizard ‘Metrics’ library](https://dropwizard.github.io/metrics/) will find that Spring Boot metrics are automatically published to com.codahale.metrics.MetricRegistry. Metrics from the MetricRegistry are also automatically exposed via the /metrics endpoint

When Dropwizard metrics are in use, the default CounterService and GaugeService are replaced with a DropwizardMetricServices, which is a wrapper around the MetricRegistry (so you can @Autowired one of those services and use it as normal). You can also create “special” Dropwizard metrics by prefixing your metric names with the appropriate type (i.e. timer.\*, histogram.\* for gauges, and meter.\* for counters).

## Message channel integration

If a MessageChannel bean called metricsChannel exists, then a MetricWriter will be created that writes metrics to that channel. Each message sent to the channel will contain a [Delta](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/metrics/writer/Delta.html) or [Metric](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/metrics/Metric.html) payload and have a metricName header. The writer is automatically hooked up to an exporter (as for all writers), so all metric values will appear on the channel, and additional analysis or actions can be taken by subscribers (it’s up to you to provide the channel and any subscribers you need).

# Auditing

Spring Boot Actuator has a flexible audit framework that will publish events once Spring Security is in play (‘authentication success’, ‘failure’ and ‘access denied’ exceptions by default). This can be very useful for reporting, and also to implement a lock-out policy based on authentication failures. To customize published security events you can provide your own implementations of AbstractAuthenticationAuditListener and AbstractAuthorizationAuditListener.

You can also choose to use the audit services for your own business events. To do that you can either inject the existing AuditEventRepository into your own components and use that directly, or you can simply publish AuditApplicationEvent via the Spring ApplicationEventPublisher (using ApplicationEventPublisherAware).

# Tracing

Tracing is automatically enabled for all HTTP requests. You can view the trace endpoint and obtain basic information about the last 100 requests:

[**{**

***"timestamp"***: 1394343677415**,**

***"info"***: **{**

***"method"***: ***"GET"*,**

***"path"***: ***"/trace"*, *"headers"***: **{**

***"request"***: **{**

***"Accept"***: ***"text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8"*, *"Connection"***: ***"keep-alive"*,**

***"Accept-Encoding"***: ***"gzip, deflate"*,**

***"User-Agent"***: ***"Mozilla/5.0 Gecko/Firefox"*, *"Accept-Language"***: ***"en-US,en;q=0.5"*,**

***"Cookie"***: ***"\_ga=GA1.1.827067509.1390890128; ..."***

***"Authorization"***: ***"Basic ..."*, *"Host"***: ***"localhost:8080"***

**},**

***"response"***: **{**

***"Strict-Transport-Security"***: ***"max-age=31536000 ; includeSubDomains"*, *"X-Application-Context"***: ***"application:8080"*,**

***"Content-Type"***: ***"application/json;charset=UTF-8"*, *"status"***: ***"200"***

**}**

**}**

**}**

},**{**

***"timestamp"***: 1394343684465**,**

...

**}]**

The following are included in the trace by default:

|  |  |
| --- | --- |
| **Name**  Request Headers | **Description**  Headers from the request. |
| Response Headers | Headers from the response. |
| Cookies | Cookie from request headers and Set-Cookie from response headers. |
| Errors | The error attributes (if any). |
| Time Taken | The time taken to service the request in milliseconds. |

## Custom tracing

If you need to trace additional events you can inject a [TraceRepository](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/trace/TraceRepository.java) into your Spring beans. The

add method accepts a single Map structure that will be converted to JSON and logged.

By default an InMemoryTraceRepository will be used that stores the last 100 events. You can define your own instance of the InMemoryTraceRepository bean if you need to expand the capacity. You can also create your own alternative TraceRepository implementation if needed.

# Process monitoring

In Spring Boot Actuator you can find a couple of classes to create files that are useful for process monitoring:

* ApplicationPidFileWriter creates a file containing the application PID (by default in the application directory with the file name application.pid).
* EmbeddedServerPortFileWriter creates a file (or files) containing the ports of the embedded server (by default in the application directory with the file name application.port).

These writers are not activated by default, but you can enable them in one of the ways described below.

## Extend configuration

In META-INF/spring.factories file you can activate the listener(s) that writes a PID file. Example:

org.springframework.context.ApplicationListener=\ org.springframework.boot.system.ApplicationPidFileWriter,\ org.springframework.boot.actuate.system.EmbeddedServerPortFileWriter

## Programmatically

You can also activate a listener by invoking the SpringApplication.addListeners(…) method and passing the appropriate Writer object. This method also allows you to customize the file name and path via the Writer constructor.

# Cloud Foundry support

Spring Boot’s actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The /cloudfoundryapplication path provides an alternative secured route to all NamedMvcEndpoint beans.

The extended support allows Cloud Foundry management UIs (such as the web application that you can use to view deployed applications) to be augmented with Spring Boot actuator information. For example, an application status page may include full health information instead of the typical “running” or “stopped” status.

**Note**

The /cloudfoundryapplication path is not directly accessible to regular users. In order to use the endpoint a valid UAA token must be passed with the request.

## Disabling extended Cloud Foundry actuator support

If you want to fully disable the /cloudfoundryapplication endpoints you can add the following to your application.properties file:

**application.properties.**

**management.cloudfoundry.enabled**=false

## Cloud Foundry self signed certificates

By default, the security verification for /cloudfoundryapplication endpoints makes SSL calls to various Cloud Foundry services. If your Cloud Foundry UAA or Cloud Controller services use self-signed certificates you will need to set the following property:

**application.properties.**

**management.cloudfoundry.skip-ssl-validation**=true

## Custom security configuration

If you define custom security configuration, and you want extended Cloud Foundry actuator support, you’ll should ensure that /cloudfoundryapplication/\*\* paths are open. Without a direct open route, your Cloud Foundry application manager will not be able to obtain endpoint data.

For Spring Security, you’ll typically include something like mvcMatchers("/ cloudfoundryapplication/\*\*").permitAll() in your configuration:

@Override

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

.authorizeRequests()

.mvcMatchers(***"/cloudfoundryapplication/\*\*"***)

.permitAll()

.mvcMatchers(***"/mypath"***)

.hasAnyRole(***"SUPERUSER"***)

.anyRequest()

.authenticated().and()

.httpBasic();

}

# What to read next

If you want to explore some of the concepts discussed in this chapter, you can take a look at the actuator [sample applications](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples). You also might want to read about graphing tools such as [Graphite](http://graphite.wikidot.com/).

Otherwise, you can continue on, to read about [‘deployment options’](#_bookmark363) or jump ahead for some in-depth information about Spring Boot’s [*build tool plugins*](#_bookmark407).

**Part VI. Deploying Spring Boot applications**

Spring Boot’s flexible packaging options provide a great deal of choice when it comes to deploying your application. You can easily deploy Spring Boot applications to a variety of cloud platforms, to a container images (such as Docker) or to virtual/real machines.

This section covers some of the more common deployment scenarios.

# Deploying to the cloud

Spring Boot’s executable jars are ready-made for most popular cloud PaaS (platform-as-a-service) providers. These providers tend to require that you “bring your own container”; they manage application processes (not Java applications specifically), so they need some intermediary layer that adapts *your* application to the *cloud’s* notion of a running process.

Two popular cloud providers, Heroku and Cloud Foundry, employ a “buildpack” approach. The buildpack wraps your deployed code in whatever is needed to *start* your application: it might be a JDK and a call to java, it might be an embedded web server, or it might be a full-fledged application server. A buildpack is pluggable, but ideally you should be able to get by with as few customizations to it as possible. This reduces the footprint of functionality that is not under your control. It minimizes divergence between development and production environments.

Ideally, your application, like a Spring Boot executable jar, has everything that it needs to run packaged within it.

In this section we’ll look at what it takes to get the [simple application that we developed](#_bookmark24) in the “Getting Started” section up and running in the Cloud.

## Cloud Foundry

Cloud Foundry provides default buildpacks that come into play if no other buildpack is specified. The Cloud Foundry [Java buildpack](https://github.com/cloudfoundry/java-buildpack) has excellent support for Spring applications, including Spring Boot. You can deploy stand-alone executable jar applications, as well as traditional .war packaged applications.

Once you’ve built your application (using, for example, mvn clean package) and [installed the cf](http://docs.cloudfoundry.org/devguide/installcf/install-go-cli.html) [command line tool](http://docs.cloudfoundry.org/devguide/installcf/install-go-cli.html), simply deploy your application using the cf push command as follows, substituting the path to your compiled .jar. Be sure to have [logged in with your cf command line client](http://docs.cloudfoundry.org/devguide/installcf/whats-new-v6.html#login) before pushing an application.

$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar

See the [cf push documentation](http://docs.cloudfoundry.org/devguide/installcf/whats-new-v6.html#push) for more options. If there is a Cloud Foundry [manifest.yml](http://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html) file present in the same directory, it will be consulted.

**Note**

Here we are substituting acloudyspringtime for whatever value you give cf as the name of your application.

At this point cf will start uploading your application:

Uploading acloudyspringtime... **OK**

Preparing to start acloudyspringtime... **OK**

-----> Downloaded app package (**8.9M**)

-----> Java Buildpack source: system

-----> Downloading Open JDK 1.7.0\_51 from .../x86\_64/openjdk-1.7.0\_51.tar.gz (**1.8s**) Expanding Open JDK to .java-buildpack/open\_jdk (**1.2s**)

-----> Downloading Spring Auto Reconfiguration from 0.8.7 .../auto-reconfiguration-0.8.7.jar (**0.1s**)

-----> Uploading droplet (**44M**)

Checking status of app 'acloudyspringtime'...

0 of 1 instances running (1 starting)

...

0 of 1 instances running (1 down)

...

0 of 1 instances running (1 starting)

...

1 of 1 instances running (1 running)

App started

Congratulations! The application is now live!

It’s easy to then verify the status of the deployed application:

$ cf apps

Getting applications in ... OK

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| name | requested state | instances | memory | disk | urls |
| ... |  |  |  |  |  |
| acloudyspringtime | started | 1/1 | 512M | 1G | acloudyspringtime.cfapps.io |
| ... |  |  |  |  |  |

Once Cloud Foundry acknowledges that your application has been deployed, you should be able to hit the application at the URI given, in this case [http://acloudyspringtime.cfapps.io/.](http://acloudyspringtime.cfapps.io/)

### Binding to services

By default, metadata about the running application as well as service connection information is exposed to the application as environment variables (for example: $VCAP\_SERVICES). This architecture decision is due to Cloud Foundry’s polyglot (any language and platform can be supported as a buildpack) nature; process-scoped environment variables are language agnostic.

Environment variables don’t always make for the easiest API so Spring Boot automatically extracts them and flattens the data into properties that can be accessed through Spring’s Environment abstraction:

@Component

**class** MyBean **implements** EnvironmentAware {

**private** String instanceId; @Override

**public void** setEnvironment(Environment environment) {

**this**.instanceId = environment.getProperty(***"vcap.application.instance\_id"***);

}

*// ...*

}

All Cloud Foundry properties are prefixed with vcap. You can use vcap properties to access application information (such as the public URL of the application) and service information (such as database credentials). See CloudFoundryVcapEnvironmentPostProcessor Javadoc for complete details.

**Tip**

The [Spring Cloud Connectors](http://cloud.spring.io/spring-cloud-connectors/) project is a better fit for tasks such as configuring a DataSource. Spring Boot includes auto-configuration support and a spring-boot-starter-cloud- connectors starter.

## Heroku

Heroku is another popular PaaS platform. To customize Heroku builds, you provide a Procfile, which provides the incantation required to deploy an application. Heroku assigns a port for the Java application to use and then ensures that routing to the external URI works.

You must configure your application to listen on the correct port. Here’s the Procfile for our starter REST application:

web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar

Spring Boot makes -D arguments available as properties accessible from a Spring Environment instance. The server.port configuration property is fed to the embedded Tomcat, Jetty or Undertow instance which then uses it when it starts up. The $PORT environment variable is assigned to us by the Heroku PaaS.

Heroku by default will use Java 1.8. This is fine as long as your Maven or Gradle build is set to use the same version (Maven users can use the java.version property). If you want to use JDK 1.7, create a new file adjacent to your pom.xml and Procfile, called system.properties. In this file add the following:

java.runtime.version=1.7

This should be everything you need. The most common workflow for Heroku deployments is to git push the code to production.

$ git push heroku master

Initializing repository, **done**. Counting objects: 95, **done**.

Delta compression using up to 8 threads. Compressing objects: 100% (78/78), **done**.

Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, **done**. Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected

-----> Installing OpenJDK 1.8... **done**

-----> Installing Maven 3.3.1... **done**

-----> Installing settings.xml... **done**

-----> Executing: mvn -B -DskipTests=true clean install

[INFO] Scanning for projects... [Downloading: http://repo.spring.io/...](http://repo.spring.io/)

[Downloaded: http://repo.spring.io/...](http://repo.spring.io/) (818 B at 1.8 KB/sec)

....

[Downloaded: http://s3pository.heroku.com/jvm/...](http://s3pository.heroku.com/jvm/) (152 KB at 595.3 KB/sec) [INFO] Installing /tmp/build\_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/... [INFO] Installing /tmp/build\_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ... [INFO] ------------------------------------------------------------------------ [INFO] **BUILD SUCCESS**

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 59.358s

[INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014 [INFO] Final Memory: 20M/493M

[INFO] ------------------------------------------------------------------------

-----> Discovering process types Procfile declares types -> **web**

-----> Compressing... **done**, 70.4MB

-----> Launching... **done**, v6

<http://agile-sierra-1405.herokuapp.com/>**deployed to Heroku**

To git@heroku.com:agile-sierra-1405.git

\* [new branch] master -> master

Your application should now be up and running on Heroku.

## OpenShift

[OpenShift](https://www.openshift.com/) is the RedHat public (and enterprise) PaaS solution. Like Heroku, it works by running scripts triggered by git commits, so you can script the launching of a Spring Boot application in pretty much any way you like as long as the Java runtime is available (which is a standard feature you can ask for at OpenShift). To do this you can use the [DIY Cartridge](https://www.openshift.com/developers/do-it-yourself) and hooks in your repository under .openshift/ action\_hooks:

The basic model is to:

1. Ensure Java and your build tool are installed remotely, e.g. using a pre\_build hook (Java and Maven are installed by default, Gradle is not)
2. Use a build hook to build your jar (using Maven or Gradle), e.g.

#!/bin/bash

cd $OPENSHIFT\_REPO\_DIR

mvn package -s .openshift/settings.xml -DskipTests=true

1. Add a start hook that calls java -jar …

#!/bin/bash

cd $OPENSHIFT\_REPO\_DIR

nohup java -jar target/\*.jar --server.port=${OPENSHIFT\_DIY\_PORT} --server.address=${OPENSHIFT\_DIY\_IP} &

1. Use a stop hook (since the start is supposed to return cleanly), e.g.

#!/bin/bash

source $OPENSHIFT\_CARTRIDGE\_SDK\_BASH

PID=$(ps -ef | grep java.\*\.jar | grep -v grep | awk '{ print $2 }') if [ -z "$PID" ]

then

client\_result "Application is already stopped"

else

kill $PID

fi

1. Embed service bindings from environment variables provided by the platform in your

application.properties, e.g.

spring.datasource.url: jdbc:mysql://${OPENSHIFT\_MYSQL\_DB\_HOST}:${OPENSHIFT\_MYSQL\_DB\_PORT}/

${OPENSHIFT\_APP\_NAME}

spring.datasource.username: ${OPENSHIFT\_MYSQL\_DB\_USERNAME} spring.datasource.password: ${OPENSHIFT\_MYSQL\_DB\_PASSWORD}

There’s a blog on [running Gradle in OpenShift](https://www.openshift.com/blogs/run-gradle-builds-on-openshift) on their website that will get you started with a gradle build to run the app.

## Amazon Web Services (AWS)

Amazon Web Services offers multiple ways to install Spring Boot based applications, either as traditional web applications (war) or as executable jar files with an embedded web server. Options include :

* AWS Elastic Beanstalk
* AWS Code Deploy
* AWS OPS Works
* AWS Cloud Formation
* AWS Container Registry

Each has different features and pricing model, here we will describe only the simplest option : AWS Elastic Beanstalk.

### AWS Elastic Beanstalk

As described in the official [Elastic Beanstalk Java guide](http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_Java.html), there are two main options to deploy a Java application; You can either use the “Tomcat Platform” or the “Java SE platform”.

#### Using the Tomcat platform

This option applies to Spring Boot projects producing a war file. There is no any special configuration required, just follow the official guide.

#### Using the Java SE platform

This option applies to Spring Boot projects producing a jar file and running an embedded web container. Elastic Beanstalk environments run an nginx instance on port 80 to proxy the actual application, running on port 5000. To configure it, add the following to your application.properties:

server.port=5000

#### Best practices

**Uploading binaries instead of sources**

By default Elastic Beanstalk uploads sources and compiles them in AWS. To upload the binaries instead, add the following to your .elasticbeanstalk/config.yml file:

deploy:

artifact: target/demo-0.0.1-SNAPSHOT.jar

**Reduce costs by setting the environment type**

By default an Elastic Beanstalk environment is load balanced. The load balancer has a cost perspective, to avoid it, set the environment type to “Single instance” as described [in the Amazon documentation](http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environments-create-wizard.html#environments-create-wizard-capacity). Single instance environments can be created using the CLI as well using the following command:

eb create -s

### Summary

This is one of the easiest ways to get to AWS, but there are more things to cover, e.g.: how to integrate Elastic Beanstalk into any CI / CD tool, using the Elastic Beanstalk maven plugin instead of the CLI, etc. There is a [blog](https://exampledriven.wordpress.com/2017/01/09/spring-boot-aws-elastic-beanstalk-example/) covering these topics more in detail.

## Boxfuse and Amazon Web Services

[Boxfuse](https://boxfuse.com/) works by turning your Spring Boot executable jar or war into a minimal VM image that can be deployed unchanged either on VirtualBox or on AWS. Boxfuse comes with deep integration for Spring

Boot and will use the information from your Spring Boot configuration file to automatically configure ports and health check URLs. Boxfuse leverages this information both for the images it produces as well as for all the resources it provisions (instances, security groups, elastic load balancers, etc).

Once you have created a [Boxfuse account](https://console.boxfuse.com/), connected it to your AWS account, and installed the latest version of the Boxfuse Client, you can deploy your Spring Boot application to AWS as follows (ensure the application has been built by Maven or Gradle first using, for example, mvn clean package):

$ boxfuse run myapp-1.0.jar -env=prod

See the [boxfuse run documentation](https://boxfuse.com/docs/commandline/run.html) for more options. If there is a [boxfuse.com/docs/commandline/](https://boxfuse.com/docs/commandline/#configuration) [#configuration](https://boxfuse.com/docs/commandline/#configuration) [boxfuse.conf] file present in the current directory, it will be consulted.

**Tip**

By default Boxfuse will activate a Spring profile named boxfuse on startup and if your executable jar or war contains an [boxfuse.com/docs/payloads/springboot.html#configuration](https://boxfuse.com/docs/payloads/springboot.html#configuration) [application-boxfuse.properties] file, Boxfuse will base its configuration based on the properties it contains.

At this point boxfuse will create an image for your application, upload it, and then configure and start the necessary resources on AWS:

Fusing Image for myapp-1.0.jar ...

Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0 Creating axelfontaine/myapp ...

Pushing axelfontaine/myapp:1.0 ... Verifying axelfontaine/myapp:1.0 ... Creating Elastic IP ...

Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...

Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1 (this may take up to 50 seconds) ...

AMI created in 00:23.557s -> ami-d23f38cf

Creating security group boxfuse-sg\_axelfontaine/myapp:1.0 ...

Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-1 ... Instance launched in 00:30.306s -> i-92ef9f53

[Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at http://52.28.235.61/](http://52.28.235.61/) ... Payload started in 00:29.266s -> <http://52.28.235.61/>

Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...

Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...

[Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at http://myapp-](http://myapp-/) axelfontaine.boxfuse.io/

Your application should now be up and running on AWS.

There’s a blog on [deploying Spring Boot apps on EC2](https://boxfuse.com/blog/spring-boot-ec2.html) as well as [documentation for the Boxfuse Spring](https://boxfuse.com/docs/payloads/springboot.html) [Boot integration](https://boxfuse.com/docs/payloads/springboot.html) on their website that will get you started with a Maven build to run the app.

## Google Cloud

Google Cloud has several options that could be used to launch Spring Boot applications. The easiest to get started with is probably App Engine, but you could also find ways to run Spring Boot in a container with Container Engine, or on a virtual machine using Compute Engine.

To run in App Engine you can create a project in the UI first, which sets up a unique identifier for you and also HTTP routes. Add a Java app to the project and leave it empty, then use the [Google Cloud](https://cloud.google.com/sdk/downloads) [SDK](https://cloud.google.com/sdk/downloads) to push your Spring Boot app into that slot from the command line or CI build.

App Engine needs you to create an app.yaml file to describe the resources your app requires. Normally you put this in src/main/appengine, and it looks something like this:

**service**: default

**runtime**: java

**env**: flex

**runtime\_config**: **jdk**: openjdk8

**handlers**:

**- url**: /.\*

**script**: this field is required**,** but ignored

**manual\_scaling**: **instances**: 1

**health\_check**: **enable\_health\_check**: False

**env\_variables**:

**ENCRYPT\_KEY**: your\_encryption\_key\_here

You can deploy the app, for example, with a Maven plugin by simply adding the project ID to the build configuration:

**<plugin>**

**<groupId>**com.google.cloud.tools**</groupId>**

**<artifactId>**appengine-maven-plugin**</artifactId>**

**<version>**1.3.0**</version>**

**<configuration>**

**<project>**myproject**</project>**

**</configuration>**

**</plugin>**

Then deploy with mvn appengine:deploy (if you need to authenticate first the build will fail).

**Note**

Google App Engine Classic is tied to the Servlet 2.5 API, so you can’t deploy a Spring Application there without some modifications. See the [Servlet 2.5 section](#_bookmark567) of this guide.

# Installing Spring Boot applications

In additional to running Spring Boot applications using java -jar it is also possible to make fully executable applications for Unix systems. A fully executable jar can be executed like any other executable binary or it can be [registered with init.d or systemd](#_bookmark379). This makes it very easy to install and manage Spring Boot applications in common production environments.

**Warning**

Fully executable jars work by embedding an extra script at the front of the file. Currently, some tools do not accept this format so you may not always be able to use this technique. For example, jar

-xf may silently fail to extract a jar or war that has been made fully-executable. It is recommended

that you only make your jar or war fully executable if you intend to execute it directly, rather than running it with java -jar or deploying it to a servlet container.

To create a ‘fully executable’ jar with Maven use the following plugin configuration:

**<plugin>**

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

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

**<configuration>**

**<executable>**true**</executable>**

**</configuration>**

**</plugin>**

With Gradle, the equivalent configuration is:

springBoot {

executable = true

}

You can then run your application by typing ./my-application.jar (where my-application is the name of your artifact). The directory containing the jar will be used as your application’s working directory.

## Supported operating systems

The default script supports most Linux distributions and is tested on CentOS and Ubuntu. Other platforms, such as OS X and FreeBSD, will require the use of a custom embeddedLaunchScript.

## Unix/Linux services

Spring Boot application can be easily started as Unix/Linux services using either init.d or systemd.

### Installation as an init.d service (System V)

If you’ve configured Spring Boot’s Maven or Gradle plugin to generate a [fully executable jar](#_bookmark377), and you’re not using a custom embeddedLaunchScript, then your application can be used as an init.d service. Simply symlink the jar to init.d to support the standard start, stop, restart and status commands.

The script supports the following features:

* Starts the services as the user that owns the jar file
* Tracks application’s PID using /var/run/<appname>/<appname>.pid
* Writes console logs to /var/log/<appname>.log

Assuming that you have a Spring Boot application installed in /var/myapp, to install a Spring Boot application as an init.d service simply create a symlink:

$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp

Once installed, you can start and stop the service in the usual way. For example, on a Debian based system:

$ service myapp start

**Tip**

If your application fails to start, check the log file written to /var/log/<appname>.log for errors.

You can also flag the application to start automatically using your standard operating system tools. For example, on Debian:

$ update-rc.d myapp defaults <priority>

#### Securing an init.d service

**Note**

The following is a set of guidelines on how to secure a Spring Boot application that’s being run as an init.d service. It is not intended to be an exhaustive list of everything that should be done to harden an application and the environment in which it runs.

When executed as root, as is the case when root is being used to start an init.d service, the default executable script will run the application as the user which owns the jar file. You should never run a Spring Boot application as root so your application’s jar file should never be owned by root. Instead, create a specific user to run your application and use chown to make it the owner of the jar file. For example:

$ chown bootapp:bootapp your-app.jar

In this case, the default executable script will run the application as the bootapp user.

**Tip**

To reduce the chances of the application’s user account being compromised, you should consider preventing it from using a login shell. Set the account’s shell to /usr/sbin/nologin, for example.

You should also take steps to prevent the modification of your application’s jar file. Firstly, configure its permissions so that it cannot be written and can only be read or executed by its owner:

$ chmod 500 your-app.jar

Secondly, you should also take steps to limit the damage if your application or the account that’s running it is compromised. If an attacker does gain access, they could make the jar file writable and change its contents. One way to protect against this is to make it immutable using chattr:

$ sudo chattr +i your-app.jar

This will prevent any user, including root, from modifying the jar.

If root is used to control the application’s service and you [use a .conf file](#_bookmark386) to customize its startup, the

.conf file will be read and evaluated by the root user. It should be secured accordingly. Use chmod so that the file can only be read by the owner and use chown to make root the owner:

$ chmod 400 your-app.conf

$ sudo chown root:root your-app.conf

### Installation as a systemd service

Systemd is the successor of the System V init system, and is now being used by many modern Linux distributions. Although you can continue to use init.d scripts with systemd, it is also possible to launch Spring Boot applications using systemd ‘service’ scripts.

Assuming that you have a Spring Boot application installed in /var/myapp, to install a Spring Boot application as a systemd service create a script named myapp.service using the following example and place it in /etc/systemd/system directory:

[Unit] Description=myapp After=syslog.target

[Service] User=myapp

ExecStart=/var/myapp/myapp.jar SuccessExitStatus=143

[Install]

WantedBy=multi-user.target

**Tip**

Remember to change the Description, User and ExecStart fields for your application.

**Tip**

Note that ExecStart field does not declare the script action command, which means that run

command is used by default.

Note that unlike when running as an init.d service, user that runs the application, PID file and console log file are managed by systemd itself and therefore must be configured using appropriate fields in ‘service’ script. Consult the [service unit configuration man page](http://www.freedesktop.org/software/systemd/man/systemd.service.html) for more details.

To flag the application to start automatically on system boot use the following command:

$ systemctl enable myapp.service

Refer to man systemctl for more details.

### Customizing the startup script

The default embedded startup script written by the Maven or Gradle plugin can be customized in a number of ways. For most people, using the default script along with a few customizations is

usually enough. If you find you can’t customize something that you need to, you can always use the

embeddedLaunchScript option to write your own file entirely.

#### Customizing script when it’s written

It often makes sense to customize elements of the start script as it’s written into the jar file. For example, init.d scripts can provide a “description” and, since you know this up front (and it won’t change), you may as well provide it when the jar is generated.

To customize written elements, use the embeddedLaunchScriptProperties option of the Spring Boot Maven or Gradle plugins.

The following property substitutions are supported with the default script:

|  |
| --- |
| **Name Description** |
| mode The script mode. Defaults to auto.  initInfoProTvhiedPersovides section of “INIT INFO”. Defaults to spring-boot-application for Gradle and to ${project.artifactId} for Maven. |
| initInfoReqTuhierReedqSutiaretd-Start section of “INIT INFO”. Defaults to $remote\_fs $syslog  $network.  initInfoReqTuhierReedqSutiorped-Stop section of “INIT INFO”. Defaults to $remote\_fs $syslog  $network. |
| initInfoDefTahuelDteSftaurltt-Start section of “INIT INFO”. Defaults to 2 3 4 5. |
| initInfoDefTahuelDteSftaouplt-Stop section of “INIT INFO”. Defaults to 0 1 6. |
| initInfoShoTrhteDSehsocrti-pDteisocnription section of “INIT INFO”. Defaults to Spring Boot Application for Gradle and to ${project.name} for Maven. |
| initInfoDesTchreiDpetsicorniption section of “INIT INFO”. Defaults to Spring Boot Application for Gradle and to ${project.description} (falling back to  ${project.name}) for Maven.  initInfoChkTchoencfhikgconfig section of “INIT INFO”. Defaults to 2345 99 01. |
| confFolder The default value for CONF\_FOLDER. Defaults to the folder containing the jar. |
| logFolder The default value for LOG\_FOLDER. Only valid for an init.d service. |
| logFilenameThe default value for LOG\_FILENAME. Only valid for an init.d service.  pidFolder The default value for PID\_FOLDER. Only valid for an init.d service. |
| pidFilenameThe default value for the name of the pid file in PID\_FOLDER. Only valid for an  init.d service.  useStartStoIpf Dthaeesmtoanrt-stop-daemon command, when it’s available, should be used to control the process. Defaults to true. |
| stopWaitTimTehe default value for STOP\_WAIT\_TIME. Only valid for an init.d service. Defaults to 60 seconds. |

#### Customizing script when it runs

For items of the script that need to be customized *after* the jar has been written you can use environment variables or a [config file](#_bookmark386).

The following environment properties are supported with the default script:

|  |
| --- |
| **Variable Description**  MODE The “mode” of operation. The default depends on the way the jar was built, but will usually be auto *(meaning it tries to guess if it is an init script by checking if it is a symlink in a directory called init.d)*. You can explicitly set it to service so that the stop|start|status|restart commands work, or to run if you just want to run the script in the foreground. |
| USE\_START\_SITf OthPe\_sDtAaErMtO-Nstop-daemon command, when it’s available, should be used to control the process. Defaults to true. |
| PID\_FOLDER The root name of the pid folder (/var/run by default).  LOG\_FOLDER The name of the folder to put log files in (/var/log by default). |
| CONF\_FOLDERThe name of the folder to read .conf files from (same folder as jar-file by default).  LOG\_FILENAMTEhe name of the log file in the LOG\_FOLDER (<appname>.log by default). |
| APP\_NAME The name of the app. If the jar is run from a symlink the script guesses the app name, but if it is not a symlink, or you want to explicitly set the app name this can be useful. |
| RUN\_ARGS The arguments to pass to the program (the Spring Boot app). |
| JAVA\_HOME The location of the java executable is discovered by using the PATH by default, but you can set it explicitly if there is an executable file at $JAVA\_HOME/bin/java. |
| JAVA\_OPTS Options that are passed to the JVM when it is launched.  JARFILE The explicit location of the jar file, in case the script is being used to launch a jar that it is not actually embedded in. |
| DEBUG if not empty will set the -x flag on the shell process, making it easy to see the logic in the script. |
| STOP\_WAIT\_TTIhMeEtime in seconds to wait when stopping the application before forcing a shutdown (60 by default). |

**Note**

The PID\_FOLDER, LOG\_FOLDER and LOG\_FILENAME variables are only valid for an init.d service. With systemd the equivalent customizations are made using ‘service’ script. Check the [service unit configuration man page](http://www.freedesktop.org/software/systemd/man/systemd.service.html) for more details.

With the exception of JARFILE and APP\_NAME, the above settings can be configured using a .conf file. The file is expected next to the jar file and have the same name but suffixed with .conf rather than .jar. For example, a jar named /var/myapp/myapp.jar will use the configuration file named

/var/myapp/myapp.conf.

**myapp.conf.**

JAVA\_OPTS=-Xmx1024M

LOG\_FOLDER=/custom/log/folder

**Tip**

You can use a CONF\_FOLDER environment variable to customize the location of the config file if you don’t like it living next to the jar.

To learn about securing this file appropriately, please refer to [the guidelines for securing an init.d service](#_bookmark381).

## Microsoft Windows services

Spring Boot application can be started as Windows service using [winsw](https://github.com/kohsuke/winsw).

A sample [maintained separately](https://github.com/snicoll-scratches/spring-boot-daemon) to the core of Spring Boot describes step-by-step how you can create a Windows service for your Spring Boot application.

# What to read next

Check out the [Cloud Foundry](http://www.cloudfoundry.com/), [Heroku](https://www.heroku.com/), [OpenShift](https://www.openshift.com/) and [Boxfuse](https://boxfuse.com/) web sites for more information about the kinds of features that a PaaS can offer. These are just four of the most popular Java PaaS providers, since Spring Boot is so amenable to cloud-based deployment you’re free to consider other providers as well.

The next section goes on to cover the [*Spring Boot CLI*](#_bookmark389); or you can jump ahead to read about [*build*](#_bookmark407)[*tool plugins*](#_bookmark407).

**Part VII. Spring Boot CLI**

The Spring Boot CLI is a command line tool that can be used if you want to quickly develop with Spring. It allows you to run Groovy scripts, which means that you have a familiar Java-like syntax, without so much boilerplate code. You can also bootstrap a new project or write your own command for it.

# Installing the CLI

The Spring Boot CLI can be installed manually; using SDKMAN! (the SDK Manager) or using Homebrew or MacPorts if you are an OSX user. See [*Section 10.2, “Installing the Spring Boot CLI”*](#_bookmark16) in the “Getting started” section for comprehensive installation instructions.

# Using the CLI

Once you have installed the CLI you can run it by typing spring. If you run spring without any arguments, a simple help screen is displayed:

$ spring

usage: spring [--help] [--version]

<command> [<args>] Available commands are:

run [options] <files> [--] [args] Run a spring groovy script

*... more command help is shown here*

You can use help to get more details about any of the supported commands. For example:

$ spring help run

spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args] Option Description

------ -----------

--autoconfigure [Boolean] Add autoconfigure compiler

transformations (default: true)

--classpath, -cp Additional classpath entries

-e, --edit Open the file with the default system editor

--no-guess-dependencies Do not attempt to guess dependencies

--no-guess-imports Do not attempt to guess imports

-q, --quiet Quiet logging

-v, --verbose Verbose logging of dependency resolution

--watch Watch the specified file for changes

The version command provides a quick way to check which version of Spring Boot you are using.

$ spring version

Spring CLI v1.5.8.RELEASE

## Running applications using the CLI

You can compile and run Groovy source code using the run command. The Spring Boot CLI is completely self-contained so you don’t need any external Groovy installation.

Here is an example “hello world” web application written in Groovy:

**hello.groovy.**

@RestController

**class** WebApplication {

@RequestMapping("/") String home() {

***"Hello World!"***

}

}

To compile and run the application type:

$ spring run hello.groovy

To pass command line arguments to the application, you need to use a -- to separate them from the “spring” command arguments, e.g.

$ spring run hello.groovy -- --server.port=9000

To set JVM command line arguments you can use the JAVA\_OPTS environment variable, e.g.

$ JAVA\_OPTS=-Xmx1024m spring run hello.groovy

### Deduced “grab” dependencies

Standard Groovy includes a @Grab annotation which allows you to declare dependencies on a third- party libraries. This useful technique allows Groovy to download jars in the same way as Maven or Gradle would, but without requiring you to use a build tool.

Spring Boot extends this technique further, and will attempt to deduce which libraries to “grab” based on your code. For example, since the WebApplication code above uses @RestController annotations, “Tomcat” and “Spring MVC” will be grabbed.

The following items are used as “grab hints”:

|  |  |
| --- | --- |
| **Items** | **Grabs** |
| JdbcTemplate, NamedParameterJdbcTemplate, DataSource | JDBC Application. |
| @EnableJms | JMS Application. |
| @EnableCaching | Caching abstraction. |
| @Test  @EnableRabbit | JUnit.  RabbitMQ. |
| @EnableReactor | Project Reactor. |
| extends Specification | Spock test. |
| @EnableBatchProcessing | Spring Batch. |
| @MessageEndpoint @EnableIntegrationPatterns | Spring Integration. |
| @EnableDeviceResolver  @Controller @RestController @EnableWebMvc | Spring Mobile.  Spring MVC + Embedded Tomcat. |
| @EnableWebSecurity  @EnableTransactionManagement | Spring Security.  Spring Transaction Management. |

**Tip**

See subclasses of [CompilerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-cli/src/main/java/org/springframework/boot/cli/compiler/CompilerAutoConfiguration.java) in the Spring Boot CLI source code to understand exactly how customizations are applied.

### Deduced “grab” coordinates

Spring Boot extends Groovy’s standard @Grab support by allowing you to specify a dependency without a group or version, for example @Grab('freemarker'). This will consult Spring Boot’s default dependency metadata to deduce the artifact’s group and version. Note that the default metadata is tied to the version of the CLI that you’re using – it will only change when you move to a new version of the CLI, putting you in control of when the versions of your dependencies may change. A table showing the dependencies and their versions that are included in the default metadata can be found in the [appendix](#_bookmark607).

### Default import statements

To help reduce the size of your Groovy code, several import statements are automatically included. Notice how the example above refers to @Component, @RestController and @RequestMapping without needing to use fully-qualified names or import statements.

**Tip**

Many Spring annotations will work without using import statements. Try running your application to see what fails before adding imports.

### Automatic main method

Unlike the equivalent Java application, you do not need to include a public static void main(String[] args) method with your Groovy scripts. A SpringApplication is automatically created, with your compiled code acting as the source.

### Custom dependency management

By default, the CLI uses the dependency management declared in spring-boot-dependencies when resolving @Grab dependencies. Additional dependency management, that will override the default dependency management, can be configured using the @DependencyManagementBom annotation. The annotation’s value should specify the coordinates (groupId:artifactId:version) of one or more Maven BOMs.

For example, the following declaration:

@DependencyManagementBom(***"com.example.custom-bom:1.0.0"***)

Will pick up custom-bom-1.0.0.pom in a Maven repository under com/example/custom- versions/1.0.0/.

When multiple BOMs are specified they are applied in the order that they’re declared. For example:

@DependencyManagementBom([***"com.example.custom-bom:1.0.0"***,

***"com.example.another-bom:1.0.0"***])

indicates that dependency management in another-bom will override the dependency management in custom-bom.

You can use @DependencyManagementBom anywhere that you can use @Grab, however, to ensure consistent ordering of the dependency management, you can only use @DependencyManagementBom at most once in your application. A useful source of dependency management (that is

a superset of Spring Boot’s dependency management) is the [Spring IO Platform](http://platform.spring.io/), e.g.

@DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE').

## Testing your code

The test command allows you to compile and run tests for your application. Typical usage looks like this:

$ spring test app.groovy tests.groovy Total: 1, Success: 1, : Failures: 0 Passed? true

In this example, tests.groovy contains JUnit @Test methods or Spock Specification classes. All the common framework annotations and static methods should be available to you without having to import them.

Here is the tests.groovy file that we used above (with a JUnit test):

**class** ApplicationTests {

@Test

**void** homeSaysHello() {

assertEquals(***"Hello World!"***, **new** WebApplication().home())

}

}

**Tip**

If you have more than one test source files, you might prefer to organize them into a test

directory.

## Applications with multiple source files

You can use “shell globbing” with all commands that accept file input. This allows you to easily use multiple files from a single directory, e.g.

$ spring run \*.groovy

This technique can also be useful if you want to segregate your “test” or “spec” code from the main application code:

$ spring test app/\*.groovy test/\*.groovy

## Packaging your application

You can use the jar command to package your application into a self-contained executable jar file. For example:

$ spring jar my-app.jar \*.groovy

The resulting jar will contain the classes produced by compiling the application and all of the application’s dependencies so that it can then be run using java -jar. The jar file will also contain entries from the application’s classpath. You can add explicit paths to the jar using --include and --exclude (both are comma-separated, and both accept prefixes to the values “+” and “-” to signify that they should be removed from the defaults). The default includes are

public/\*\*, resources/\*\*, static/\*\*, templates/\*\*, META-INF/\*\*, \*

and the default excludes are

.\*, repository/\*\*, build/\*\*, target/\*\*, \*\*/\*.jar, \*\*/\*.groovy

See the output of spring help jar for more information.

## Initialize a new project

The init command allows you to create a new project using [start.spring.io](https://start.spring.io/) without leaving the shell. For example:

$ spring init --dependencies=web,data-jpa my-project Using service at https://start.spring.io

Project extracted to '/Users/developer/example/my-project'

This creates a my-project directory with a Maven-based project using spring-boot-starter- web and spring-boot-starter-data-jpa. You can list the capabilities of the service using the -- list flag

$ spring init --list

=======================================

Capabilities of https://start.spring.io

=======================================

Available dependencies:

-----------------------

actuator - Actuator: Production ready features to help you monitor and manage your application

...

web - Web: Support for full-stack web development, including Tomcat and spring-webmvc websocket - Websocket: Support for WebSocket development

ws - WS: Support for Spring Web Services

Available project types:

------------------------

gradle-build - Gradle Config [format:build, build:gradle] gradle-project - Gradle Project [format:project, build:gradle] maven-build - Maven POM [format:build, build:maven]

maven-project - Maven Project [format:project, build:maven] (default)

...

The init command supports many options, check the help output for more details. For instance, the following command creates a gradle project using Java 8 and war packaging:

$ spring init --build=gradle --java-version=1.8 --dependencies=websocket --packaging=war sample-app.zip Using service at https://start.spring.io

Content saved to 'sample-app.zip'

## Using the embedded shell

Spring Boot includes command-line completion scripts for BASH and zsh shells. If you don’t use either of these shells (perhaps you are a Windows user) then you can use the shell command to launch an integrated shell.

$ spring shell

**Spring Boot** (v1.5.8.RELEASE)

Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.

From inside the embedded shell you can run other commands directly:

$ version

Spring CLI v1.5.8.RELEASE

The embedded shell supports ANSI color output as well as tab completion. If you need to run a native command you can use the ! prefix. Hitting ctrl-c will exit the embedded shell.

## Adding extensions to the CLI

You can add extensions to the CLI using the install command. The command takes one or more sets of artifact coordinates in the format group:artifact:version. For example:

$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE

In addition to installing the artifacts identified by the coordinates you supply, all of the artifacts' dependencies will also be installed.

To uninstall a dependency use the uninstall command. As with the install command, it takes one or more sets of artifact coordinates in the format group:artifact:version. For example:

$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE

It will uninstall the artifacts identified by the coordinates you supply and their dependencies. To uninstall all additional dependencies you can use the --all option. For example:

$ spring uninstall --all

# Developing application with the Groovy beans DSL

Spring Framework 4.0 has native support for a beans{} “DSL” (borrowed from [Grails](http://grails.org/)), and you can embed bean definitions in your Groovy application scripts using the same format. This is sometimes a good way to include external features like middleware declarations. For example:

@Configuration

**class** Application **implements** CommandLineRunner {

@Autowired SharedService service

@Override

**void** run(String... args) { println service.message

}

}

**import** my.company.SharedService beans {

service(SharedService) { message = ***"Hello World"***

}

}

You can mix class declarations with beans{} in the same file as long as they stay at the top level, or you can put the beans DSL in a separate file if you prefer.

# Configuring the CLI with settings.xml

The Spring Boot CLI uses Aether, Maven’s dependency resolution engine, to resolve dependencies. The CLI makes use of the Maven configuration found in ~/.m2/settings.xml to configure Aether. The following configuration settings are honored by the CLI:

* Offline
* Mirrors
* Servers
* Proxies
* Profiles
  + Activation
  + Repositories
* Active profiles

Please refer to [Maven’s settings documentation](https://maven.apache.org/settings.html) for further information.

# What to read next

There are some [sample groovy scripts](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-cli/samples) available from the GitHub repository that you can use to try out the Spring Boot CLI. There is also extensive Javadoc throughout the [source code](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-cli/src/main/java/org/springframework/boot/cli).

If you find that you reach the limit of the CLI tool, you will probably want to look at converting your application to full Gradle or Maven built “groovy project”. The next section covers Spring Boot’s [*Build*](#_bookmark407)[*tool plugins*](#_bookmark407) that you can use with Gradle or Maven.

**Part VIII. Build tool plugins**

Spring Boot provides build tool plugins for Maven and Gradle. The plugins offer a variety of features, including the packaging of executable jars. This section provides more details on both plugins, as well as some help should you need to extend an unsupported build system. If you are just getting started, you might want to read “[Chapter 13, *Build systems*](#_bookmark35)” from the [Part III, “Using Spring Boot”](#_bookmark34) section first.

# Spring Boot Maven plugin

The [Spring Boot Maven Plugin](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/) provides Spring Boot support in Maven, allowing you to package executable jar or war archives and run an application “in-place”. To use it you must be using Maven

3.2 (or better).

**Note**

Refer to the [Spring Boot Maven Plugin Site](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/) for complete plugin documentation.

## Including the plugin

To use the Spring Boot Maven Plugin simply include the appropriate XML in the plugins section of your pom.xml

<?xml version="1.0" encoding="UTF-8"?>

**<project xmlns**=[**"http://maven.apache.org/POM/4.0.0"**](http://maven.apache.org/POM/4.0.0) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://maven.apache.org/POM/4.0.0**](http://maven.apache.org/POM/4.0.0)[**http://maven.apache.org/xsd/maven-4.0.0.xsd"**](http://maven.apache.org/xsd/maven-4.0.0.xsd)**>**

**<modelVersion>**4.0.0**</modelVersion>**

*<!-- ... -->*

**<build>**

**<plugins>**

**<plugin>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**<executions>**

**<execution>**

**<goals>**

**<goal>**repackage**</goal>**

**</goals>**

**</execution>**

**</executions>**

**</plugin>**

**</plugins>**

**</build>**

**</project>**

This configuration will repackage a jar or war that is built during the package phase of the Maven lifecycle. The following example shows both the repackaged jar, as well as the original jar, in the target directory:

$ mvn package

$ ls target/\*.jar

target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original

If you don’t include the <execution/> configuration as above, you can run the plugin on its own (but only if the package goal is used as well). For example:

$ mvn package spring-boot:repackage

$ ls target/\*.jar

target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original

If you are using a milestone or snapshot release you will also need to add appropriate

pluginRepository elements:

**<pluginRepositories>**

**<pluginRepository>**

**<id>**spring-snapshots**</id>**

**<url>**<http://repo.spring.io/snapshot>**</url>**

**</pluginRepository>**

**<pluginRepository>**

**<id>**spring-milestones**</id>**

**<url>**<http://repo.spring.io/milestone>**</url>**

**</pluginRepository>**

**</pluginRepositories>**

## Packaging executable jar and war files

Once spring-boot-maven-plugin has been included in your pom.xml it will automatically attempt to rewrite archives to make them executable using the spring-boot:repackage goal. You should configure your project to build a jar or war (as appropriate) using the usual packaging element:

<?xml version="1.0" encoding="UTF-8"?>

**<project xmlns**=[**"http://maven.apache.org/POM/4.0.0"**](http://maven.apache.org/POM/4.0.0) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://maven.apache.org/POM/4.0.0**](http://maven.apache.org/POM/4.0.0)[**http://maven.apache.org/xsd/maven-4.0.0.xsd"**](http://maven.apache.org/xsd/maven-4.0.0.xsd)**>**

*<!-- ... -->*

**<packaging>**jar**</packaging>**

*<!-- ... -->*

**</project>**

Your existing archive will be enhanced by Spring Boot during the package phase. The main class that you want to launch can either be specified using a configuration option, or by adding a Main-Class attribute to the manifest in the usual way. If you don’t specify a main class the plugin will search for a class with a public static void main(String[] args) method.

To build and run a project artifact, you can type the following:

$ mvn package

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

To build a war file that is both executable and deployable into an external container you need to mark the embedded container dependencies as “provided”, e.g:

<?xml version="1.0" encoding="UTF-8"?>

**<project xmlns**=[**"http://maven.apache.org/POM/4.0.0"**](http://maven.apache.org/POM/4.0.0) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://maven.apache.org/POM/4.0.0**](http://maven.apache.org/POM/4.0.0)[**http://maven.apache.org/xsd/maven-4.0.0.xsd"**](http://maven.apache.org/xsd/maven-4.0.0.xsd)**>**

*<!-- ... -->*

**<packaging>**war**</packaging>**

*<!-- ... -->*

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

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

**<scope>**provided**</scope>**

**</dependency>**

*<!-- ... -->*

**</dependencies>**

**</project>**

**Tip**

See the “[Section 85.1, “Create a deployable war file”](#_bookmark563)” section for more details on how to create a deployable war file.

Advanced configuration options and examples are available in the [plugin info page](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/).

# Spring Boot Gradle plugin

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, allowing you to package executable jar or war archives, run Spring Boot applications and use the dependency management provided by spring-boot-dependencies.

## Including the plugin

To use the Spring Boot Gradle Plugin configure it using the plugins block:

plugins {

id ***'org.springframework.boot'*** version ***'1.5.8.RELEASE'***

}

## Gradle dependency management

The spring-boot plugin automatically applies the [Dependency Management Plugin](https://github.com/spring-gradle-plugins/dependency-management-plugin/) and configures it to import the spring-boot-starter-parent bom. This provides a similar dependency management experience to the one that is enjoyed by Maven users. For example, it allows you to omit version numbers when declaring dependencies that are managed in the bom. To make use of this functionality, simply declare dependencies in the usual way, but leave the version number empty:

dependencies {

compile(***"org.springframework.boot:spring-boot-starter-web"***) compile(***"org.thymeleaf:thymeleaf-spring4"***) compile(***"nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect"***)

}

**Note**

The version of the spring-boot gradle plugin that you declare determines the version of the spring-boot-starter-parent bom that is imported (this ensures that builds are always repeatable). You should always set the version of the spring-boot gradle plugin to the actual Spring Boot version that you wish to use. Details of the versions that are provided can be found in the [appendix](#_bookmark607).

To learn more about the capabilities of the Dependency Management Plugin, please refer to its [documentation](https://github.com/spring-gradle-plugins/dependency-management-plugin/blob/master/README.md).

## Packaging executable jar and war files

Once the spring-boot plugin has been applied to your project it will automatically attempt to rewrite archives to make them executable using the [bootRepackage task](#_bookmark417). You should configure your project to build a jar or war (as appropriate) in the usual way.

The main class that you want to launch can either be specified using a configuration option, or by adding a Main-Class attribute to the manifest. If you don’t specify a main class the plugin will search for a class with a public static void main(String[] args) method.

**Tip**

Check [Section 67.6, “Repackage configuration”](#_bookmark417) for a full list of configuration options.

To build and run a project artifact, you can type the following:

$ gradle build

$ java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar

To build a war file that is both executable and deployable into an external container, you need to mark the embedded container dependencies as belonging to the war plugin’s providedRuntime configuration, e.g.:

...

apply plugin: ***'war'***

war {

baseName = ***'myapp'***

version = ***'0.5.0'***

}

repositories { jcenter()

maven { url [***"http://repo.spring.io/libs-snapshot"***](http://repo.spring.io/libs-snapshot) }

}

dependencies {

compile(***"org.springframework.boot:spring-boot-starter-web"***) providedRuntime(***"org.springframework.boot:spring-boot-starter-tomcat"***)

...

}

**Tip**

See the “[Section 85.1, “Create a deployable war file”](#_bookmark563)” section for more details on how to create a deployable war file.

## Running a project in-place

To run a project in place without building a jar first you can use the “bootRun” task:

$ gradle bootRun

If [devtools](#_bookmark67) has been added to your project it will automatically monitor your application for changes. Alternatively, you can also run the application so that your static classpath resources (i.e. in src/main/ resources by default) are reloadable in the live application, which can be helpful at development time.

bootRun {

addResources = true

}

Making static classpath resources reloadable means that bootRun does not use the output of the processResources task, i.e., when invoked using bootRun, your application will use the resources in their unprocessed form.

## Spring Boot plugin configuration

The gradle plugin automatically extends your build script DSL with a springBoot element for global configuration of the Boot plugin. Set the appropriate properties as you would with any other Gradle extension (see below for a list of configuration options):

springBoot {

backupSource = false

}

## Repackage configuration

The plugin adds a bootRepackage task which you can also configure directly, e.g.:

bootRepackage {

mainClass = ***'demo.Application'***

}

The following configuration options are available:

|  |  |
| --- | --- |
| **Name** | **Description** |
| enabled  mainClass | Boolean flag to switch the repackager off (sometimes useful if you want the other Boot features but not this one)  The main class that should be run. If not specified, and you have applied the application plugin, the mainClassName project property will be used. If the application plugin has not been applied or no mainClassName has been specified, the archive will be searched for a suitable class. "Suitable" means a unique class with a well-formed main() method (if more than one is found the build will fail). If you have applied the application plugin, the main class can also be specified via its "run" task (main property) and/or its "startScripts" task (mainClassName property) as an alternative to using the "springBoot" configuration. |
| classifier | A file name segment (before the extension) to add to the archive, so that the original is preserved in its original location. Defaults to null in which case the archive is repackaged in place. The default is convenient for many purposes, but if you want to use  the original jar as a dependency in another project you must use a classifier to define the executable archive. |
| withJarTask | The name or value of the Jar task (defaults to all tasks of type  Jar) which is used to locate the archive to repackage. |
| customConfiguration  executable | The name of the custom configuration which is used to populate the nested lib directory (without specifying this you get all compile and runtime dependencies).  Boolean flag to indicate if jar files are fully executable on Unix like operating systems. Defaults to false. |
| embeddedLaunchScript  embeddedLaunchScriptProp | The embedded launch script to prepend to the front of the jar if it is fully executable. If not specified the 'Spring Boot' default script will be used.  erAtdideitisonal properties that to be expanded in the launch script. The default script supports a mode property which can contain the values auto, service or run. |
| excludeDevtools | Boolean flag to indicate if the devtools jar should be excluded from the repackaged archives. Defaults to true. |

## Repackage with custom Gradle configuration

Sometimes it may be more appropriate to not package default dependencies resolved from compile, runtime and provided scopes. If the created executable jar file is intended to be run as it is, you need to have all dependencies nested inside it; however, if the plan is to explode a jar file and run the main class manually, you may already have some of the libraries available via CLASSPATH. This is a situation where you can repackage your jar with a different set of dependencies.

Using a custom configuration will automatically disable dependency resolving from compile, runtime and provided scopes. Custom configuration can be either defined globally (inside the springBoot section) or per task.

task clientJar(type: Jar) { appendix = ***'client'***

from sourceSets.main.output exclude(***'\*\*/\*Something\*'***)

}

task clientBoot(type: BootRepackage, dependsOn: clientJar) { withJarTask = clientJar

customConfiguration = ***"mycustomconfiguration"***

}

In above example, we created a new clientJar Jar task to package a customized file set from your compiled sources. Then we created a new clientBoot BootRepackage task and instructed it to work with only clientJar task and mycustomconfiguration.

configurations {

mycustomconfiguration.exclude group: ***'log4j'***

}

dependencies {

mycustomconfiguration configurations.runtime

}

The configuration that we are referring to in BootRepackage is a normal [Gradle configuration](https://docs.gradle.org/2.14.1/dsl/org.gradle.api.artifacts.Configuration.html). In the above example we created a new configuration named mycustomconfiguration instructing it to derive from a runtime and exclude the log4j group. If the clientBoot task is executed, the repackaged boot jar will have all dependencies from runtime but no log4j jars.

### Configuration options

The following configuration options are available:

|  |  |
| --- | --- |
| **Name**  mainClass | **Description**  The main class that should be run by the executable archive. |
| providedConfiguration  backupSource | The name of the provided configuration (defaults to  providedRuntime).  If the original source archive should be backed-up before being repackaged (defaults to true). |
| customConfiguration layout | The name of the custom configuration.  The type of archive, corresponding to how the dependencies are laid out inside (defaults to a guess based on the archive type).  See [available layouts for more details](#_bookmark420). |

|  |  |
| --- | --- |
| **Name** | **Description** |
| layoutFactory  requiresUnpack | A layout factory that can be used if a custom layout is required. Alternative layouts can be provided by 3rd parties. Layout factories are only used when layout is not specified.  A list of dependencies (in the form “groupId:artifactId” that must be unpacked from fat jars in order to run. Items are still packaged into the fat jar, but they will be automatically unpacked when it runs. |

### Available layouts

The layout attribute configures the format of the archive and whether the bootstrap loader should be included or not. The following layouts are available:

|  |  |  |
| --- | --- | --- |
| **Name**  JAR | **Description**  Regular executable [JAR layout](#_bookmark595). | **Executable**  Yes |
| WAR | Executable [WAR layout](#_bookmark596). provided dependencies are placed in WEB-INF/lib-provided to avoid any clash when the war is deployed in a servlet container. | Yes |
| ZIP (alias to DIR) MODULE | Similar to JAR layout, using [PropertiesLauncher](#_bookmark602).  Bundle dependencies (excluding those with provided  scope) and project resources. | Yes No |
| NONE | Bundle all dependencies and project resources. | No |

### Using a custom layout

If you have custom requirements for how to arrange the dependencies and loader classes inside the repackaged jar, you can use a custom layout. Any library which defines one or more LayoutFactory implementations can be added to the build script dependencies and then the layout factory becomes available in the springBoot configuration. For example:

buildscript { repositories {

mavenCentral()

}

dependencies {

classpath(***"org.springframework.boot:spring-boot-gradle-plugin:1.5.8.RELEASE"***) classpath(***"com.example:custom-layout:1.0.0"***)

}

}

springBoot {

layoutFactory = **new** com.example.CustomLayoutFactory()

}

**Note**

If there is only one custom LayoutFactory on the build classpath and it is listed in META-INF/ spring.factories then it is unnecessary to explicitly set it in the springBoot configuration. Layout factories are only used when no explicit layout is specified.

## Understanding how the Gradle plugin works

When spring-boot is applied to your Gradle project a default task named bootRepackage is created automatically. The bootRepackage task depends on Gradle assemble task, and when executed, it tries to find all jar artifacts whose qualifier is empty (i.e. tests and sources jars are automatically skipped).

Due to the fact that bootRepackage finds 'all' created jar artifacts, the order of Gradle task execution is important. Most projects only create a single jar file, so usually this is not an issue; however, if you are planning to create a more complex project setup, with custom Jar and BootRepackage tasks, there are few tweaks to consider.

If you are 'just' creating custom jar files from your project you can simply disable default jar and

bootRepackage tasks:

jar.enabled = false bootRepackage.enabled = false

Another option is to instruct the default bootRepackage task to only work with a default jar task.

bootRepackage.withJarTask = jar

If you have a default project setup where the main jar file is created and repackaged, 'and' you still want to create additional custom jars, you can combine your custom repackage tasks together and use dependsOn so that the bootJars task will run after the default bootRepackage task is executed:

task bootJars

bootJars.dependsOn = [clientBoot1,clientBoot2,clientBoot3] build.dependsOn(bootJars)

All the above tweaks are usually used to avoid situations where an already created boot jar is repackaged again. Repackaging an existing boot jar will not break anything, but you may find that it includes unnecessary dependencies.

## Publishing artifacts to a Maven repository using Gradle

If you are declaring dependencies without versions and you want to publish artifacts to a Maven repository you will need to configure the Maven publication with details of Spring Boot’s dependency management. This can be achieved by configuring it to publish poms that inherit from spring-boot- starter-parent or that import dependency management from spring-boot-dependencies. The exact details of this configuration depend on how you’re using Gradle and how you’re trying to publish the artifacts.

### Configuring Gradle to produce a pom that inherits dependency management

The following is an example of configuring Gradle to generate a pom that inherits from spring-boot- starter-parent. Please refer to the [Gradle User Guide](https://docs.gradle.org/2.14.1/userguide/userguide.html) for further information.

uploadArchives { repositories {

mavenDeployer { pom {

project {

parent {

groupId ***"org.springframework.boot"***

artifactId ***"spring-boot-starter-parent"***

version ***"1.5.8.RELEASE"***

}

}

}

}

}

}

### Configuring Gradle to produce a pom that imports dependency management

The following is an example of configuring Gradle to generate a pom that imports the dependency management provided by spring-boot-dependencies. Please refer to the [Gradle User Guide](http://www.gradle.org/docs/current/userguide/userguide.html) for further information.

uploadArchives { repositories {

mavenDeployer { pom {

project {

dependencyManagement { dependencies {

dependency {

groupId ***"org.springframework.boot"*** artifactId ***"spring-boot-dependencies"*** version ***"1.5.8.RELEASE"***

type ***"pom"***

scope ***"import"***

}

}

}

}

}

}

}

}

# Spring Boot AntLib module

The Spring Boot AntLib module provides basic Spring Boot support for Apache Ant. You can use the module to create executable jars. To use the module you need to declare an additional spring-boot namespace in your build.xml:

**<project xmlns:ivy**=**"antlib:org.apache.ivy.ant" xmlns:spring-boot**=**"antlib:org.springframework.boot.ant" name**=**"myapp" default**=**"build">**

...

**</project>**

You’ll need to remember to start Ant using the -lib option, for example:

$ ant -lib <folder containing spring-boot-antlib-1.5.8.RELEASE.jar>

**Tip**

The “Using Spring Boot” section includes a more complete example of [using Apache Ant with](#_bookmark43)

[spring-boot-antlib](#_bookmark43)

## Spring Boot Ant tasks

Once the spring-boot-antlib namespace has been declared, the following additional tasks are available.

### spring-boot:exejar

The exejar task can be used to creates a Spring Boot executable jar. The following attributes are supported by the task:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| destfile | The destination jar file to create | Yes |
| classes start-class | The root directory of Java class files The main application class to run | Yes  No *(default is first class found declaring a main method)* |

The following nested elements can be used with the task:

|  |
| --- |
| **Element Description** |
| resources One or more [Resource Collections](http://ant.apache.org/manual/Types/resources.html#collection) describing a set of [Resources](http://ant.apache.org/manual/Types/resources.html) that should be added to the content of the created jar file.  lib One or more [Resource Collections](http://ant.apache.org/manual/Types/resources.html#collection) that should be added to the set of jar libraries that make up the runtime dependency classpath of the application. |

### Examples

**Specify start-class.**

**<spring-boot:exejar destfile**=**"target/my-application.jar" classes**=**"target/classes" start-class**=**"com.foo.MyApplication">**

**<resources>**

**<fileset dir**=**"src/main/resources" />**

**</resources>**

**<lib>**

**<fileset dir**=**"lib" />**

**</lib>**

**</spring-boot:exejar>**

**Detect start-class.**

**<exejar destfile**=**"target/my-application.jar" classes**=**"target/classes">**

**<lib>**

**<fileset dir**=**"lib" />**

**</lib>**

**</exejar>**

## spring-boot:findmainclass

The findmainclass task is used internally by exejar to locate a class declaring a main. You can also use this task directly in your build if needed. The following attributes are supported

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| classesroot | The root directory of Java class files | Yes *(unless mainclass is specified)* |
| mainclass | Can be used to short-circuit the main | No |
|  | class search |  |
| property | The Ant property that should be set | No *(result will be logged if unspecified)* |
|  | with the result |  |

### Examples

**Find and log.**

**<findmainclass classesroot**=**"target/classes" />**

**Find and set.**

**<findmainclass classesroot**=**"target/classes" property**=**"main-class" />**

**Override and set.**

**<findmainclass mainclass**=**"com.foo.MainClass" property**=**"main-class" />**

# Supporting other build systems

If you want to use a build tool other than Maven, Gradle or Ant, you will likely need to develop your own plugin. Executable jars need to follow a specific format and certain entries need to be written in an uncompressed form (see the [*executable jar format*](#_bookmark593) section in the appendix for details).

The Spring Boot Maven and Gradle plugins both make use of spring-boot-loader-tools to actually generate jars. You are also free to use this library directly yourself if you need to.

## Repackaging archives

To repackage an existing archive so that it becomes a self-contained executable archive use org.springframework.boot.loader.tools.Repackager. The Repackager class takes a single constructor argument that refers to an existing jar or war archive. Use one of the two available repackage() methods to either replace the original file or write to a new destination. Various settings can also be configured on the repackager before it is run.

## Nested libraries

When repackaging an archive you can include references to dependency files using the org.springframework.boot.loader.tools.Libraries interface. We don’t provide any concrete implementations of Libraries here as they are usually build system specific.

If your archive already includes libraries you can use Libraries.NONE.

## Finding a main class

If you don’t use Repackager.setMainClass() to specify a main class, the repackager will use [ASM](http://asm.ow2.org/) to read class files and attempt to find a suitable class with a public static void main(String[] args) method. An exception is thrown if more than one candidate is found.

## Example repackage implementation

Here is a typical example repackage:

Repackager repackager = **new** Repackager(sourceJarFile); repackager.setBackupSource(false); repackager.repackage(**new** Libraries() {

@Override

**public void** doWithLibraries(LibraryCallback callback) **throws** IOException {

*// Build system specific implementation, callback for each dependency*

*// callback.library(new Library(nestedFile, LibraryScope.COMPILE));*

}

});

# What to read next

If you’re interested in how the build tool plugins work you can look at the [spring-boot-tools](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-tools) module on GitHub. More technical details of the [executable jar format](#_bookmark593) are covered in the appendix.

If you have specific build-related questions you can check out the “[how-to](#_bookmark438)” guides.

**Part IX. ‘How-to’ guides**

This section provides answers to some common ‘how do I do that…’ type of questions that often arise when using Spring Boot. This is by no means an exhaustive list, but it does cover quite a lot.

If you are having a specific problem that we don’t cover here, you might want to check out [stackoverflow.com](http://stackoverflow.com/tags/spring-boot) to see if someone has already provided an answer; this is also a great place to ask new questions (please use the spring-boot tag).

We’re also more than happy to extend this section; If you want to add a ‘how-to’ you can send us a [pull request](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE).

# Spring Boot application

## Create your own FailureAnalyzer

[FailureAnalyzer](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/diagnostics/FailureAnalyzer.html) is a great way to intercept an exception on startup and turn it into a human-readable message, wrapped into a [FailureAnalysis](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/diagnostics/FailureAnalysis.html). Spring Boot provides such analyzer for application context related exceptions, JSR-303 validations and more. It is actually very easy to create your own.

AbstractFailureAnalyzer is a convenient extension of FailureAnalyzer that checks the presence of a specified exception type in the exception to handle. You can extend from that so that your implementation gets a chance to handle the exception only when it is actually present. If for whatever reason you can’t handle the exception, return null to give another implementation a chance to handle the exception.

FailureAnalyzer implementations are to be registered in a META-INF/spring.factories: the following registers ProjectConstraintViolationFailureAnalyzer:

**org.springframework.boot.diagnostics.FailureAnalyzer**=\ com.example.ProjectConstraintViolationFailureAnalyzer

## Troubleshoot auto-configuration

The Spring Boot auto-configuration tries its best to ‘do the right thing’, but sometimes things fail and it can be hard to tell why.

There is a really useful ConditionEvaluationReport available in any Spring Boot ApplicationContext. You will see it if you enable DEBUG logging output. If you use the spring- boot-actuator there is also an autoconfig endpoint that renders the report in JSON. Use that to debug the application and see what features have been added (and which not) by Spring Boot at runtime.

Many more questions can be answered by looking at the source code and the Javadoc. Some rules of thumb:

* Look for classes called \*AutoConfiguration and read their sources, in particular the @Conditional\* annotations to find out what features they enable and when. Add --debug to the command line or a System property -Ddebug to get a log on the console of all the auto-configuration decisions that were made in your app. In a running Actuator app look at the autoconfig endpoint (‘/autoconfig’ or the JMX equivalent) for the same information.
* Look for classes that are @ConfigurationProperties (e.g. [ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java)) and read from there the available external configuration options. The @ConfigurationProperties has a name attribute which acts as a prefix to external properties, thus ServerProperties has prefix="server" and its configuration properties are server.port, server.address etc. In a running Actuator app look at the configprops endpoint.
* Look for use of RelaxedPropertyResolver to pull configuration values explicitly out of the

Environment. It often is used with a prefix.

* Look for @Value annotations that bind directly to the Environment. This is less flexible than the RelaxedPropertyResolver approach, but does allow some relaxed binding, specifically for OS environment variables (so CAPITALS\_AND\_UNDERSCORES are synonyms for period.separated).
* Look for @ConditionalOnExpression annotations that switch features on and off in response to SpEL expressions, normally evaluated with placeholders resolved from the Environment.

## Customize the Environment or ApplicationContext before it starts

A SpringApplication has ApplicationListeners and ApplicationContextInitializers that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from META-INF/spring.factories. There is more than one way to register additional ones:

* Programmatically per application by calling the addListeners and addInitializers methods on SpringApplication before you run it.
* Declaratively per application by setting context.initializer.classes or

context.listener.classes.

* Declaratively for all applications by adding a META-INF/spring.factories and packaging a jar file that the applications all use as a library.

The SpringApplication sends some special ApplicationEvents to the listeners (even some before the context is created), and then registers the listeners for events published by the ApplicationContext as well. See [*Section 23.5, “Application events and listeners”*](#_bookmark91) in the ‘Spring Boot features’ section for a complete list.

It is also possible to customize the Environment before the application context is refreshed using EnvironmentPostProcessor. Each implementation should be registered in META-INF/ spring.factories:

**org.springframework.boot.env.EnvironmentPostProcessor**=com.example.YourEnvironmentPostProcessor

The implementation can load arbitrary files and add them to the Environment. For instance, this example loads a YAML configuration file from the classpath:

**public class** EnvironmentPostProcessorExample **implements** EnvironmentPostProcessor {

**private final** YamlPropertySourceLoader loader = **new** YamlPropertySourceLoader(); @Override

**public void** postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application) {

Resource path = **new** ClassPathResource(***"com/example/myapp/config.yml"***); PropertySource<?> propertySource = loadYaml(path); environment.getPropertySources().addLast(propertySource);

}

**private** PropertySource<?> loadYaml(Resource path) {

**if** (!path.exists()) {

**throw new** IllegalArgumentException(***"Resource "*** + path + ***" does not exist"***);

}

**try** {

**return this**.loader.load(***"custom-resource"***, path, null);

}

**catch** (IOException ex) {

**throw new** IllegalStateException(

***"Failed to load yaml configuration from "*** + path, ex);

}

}

}

**Tip**

The Environment will already have been prepared with all the usual property sources that Spring Boot loads by default. It is therefore possible to get the location of the file from the environment. This example adds the custom-resource property source at the end of the list so that a key defined in any of the usual other locations takes precedence. A custom implementation may obviously defines another order.

**Note**

While using @PropertySource on your @SpringBootApplication seems convenient and easy enough to load a custom resource in the Environment, we do not recommend it as Spring Boot prepares the Environment before the ApplicationContext is refreshed. Any key defined via @PropertySource will be loaded too late to have any effect on auto-configuration.

## Build an ApplicationContext hierarchy (adding a parent or root context)

You can use the ApplicationBuilder class to create parent/child ApplicationContext hierarchies. See [*Section 23.4, “Fluent builder API”*](#_bookmark90)in the ‘Spring Boot features’ section for more information.

## Create a non-web application

Not all Spring applications have to be web applications (or web services). If you want to execute some code in a main method, but also bootstrap a Spring application to set up the infrastructure to use, then it’s easy with the SpringApplication features of Spring Boot. A SpringApplication changes its ApplicationContext class depending on whether it thinks it needs a web application or not. The first thing you can do to help it is to just leave the servlet API dependencies off the classpath. If you can’t do that (e.g. you are running 2 applications from the same code base) then you can explicitly call setWebEnvironment(false) on your SpringApplication instance, or set the applicationContextClass property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a CommandLineRunner and dropped into the context as a @Bean definition.

# Properties & configuration

## Automatically expand properties at build time

Rather than hardcoding some properties that are also specified in your project’s build configuration, you can automatically expand them using the existing build configuration instead. This is possible in both Maven and Gradle.

### Automatic property expansion using Maven

You can automatically expand properties from the Maven project using resource filtering. If you use the spring-boot-starter-parent you can then refer to your Maven ‘project properties’ via @..@ placeholders, e.g.

[**app.encoding**=@project.build.sourceEncoding@](mailto:app.encoding%3D@project.build.sourceEncoding@) [**app.java.version**=@java.version@](mailto:app.java.version%3D@java.version@)

**Note**

Only production configuration is filtered that way (i.e. no filtering is applied on src/test/ resources).

**Tip**

The spring-boot:run can add src/main/resources directly to the classpath (for hot reloading purposes) if you enable the addResources flag. This circumvents the resource filtering and this feature. You can use the exec:java goal instead or customize the plugin’s configuration, see the [plugin usage page](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/usage.html) for more details.

If you don’t use the starter parent, in your pom.xml you need (inside the <build/> element):

**<resources>**

**<resource>**

**<directory>**src/main/resources**</directory>**

**<filtering>**true**</filtering>**

**</resource>**

**</resources>**

and (inside <plugins/>):

**<plugin>**

**<groupId>**org.apache.maven.plugins**</groupId>**

**<artifactId>**maven-resources-plugin**</artifactId>**

**<version>**2.7**</version>**

**<configuration>**

**<delimiters>**

**<delimiter>**@**</delimiter>**

**</delimiters>**

**<useDefaultDelimiters>**false**</useDefaultDelimiters>**

**</configuration>**

**</plugin>**

**Note**

The useDefaultDelimiters property is important if you are using standard Spring placeholders in your configuration (e.g. ${foo}). These may be expanded by the build if that property is not set to false.

### Automatic property expansion using Gradle

You can automatically expand properties from the Gradle project by configuring the Java plugin’s

processResources task to do so:

processResources { expand(project.properties)

}

You can then refer to your Gradle project’s properties via placeholders, e.g.

**app.name**=${name} **app.description**=${description}

**Note**

Gradle’s expand method uses Groovy’s SimpleTemplateEngine which transforms ${..} tokens. The ${..} style conflicts with Spring’s own property placeholder mechanism. To use Spring property placeholders together with automatic expansion the Spring property placeholders need to be escaped like \${..}.

## Externalize the configuration of SpringApplication

A SpringApplication has bean properties (mainly setters) so you can use its Java API as you create the application to modify its behavior. Or you can externalize the configuration using properties in spring.main.\*. E.g. in application.properties you might have.

**spring.main.web-environment**=false **spring.main.banner-mode**=off

and then the Spring Boot banner will not be printed on startup, and the application will not be a web application.

**Note**

The example above also demonstrates how flexible binding allows the use of underscores (\_) as well as dashes (-) in property names.

Properties defined in external configuration overrides the values specified via the Java API with the notable exception of the sources used to create the ApplicationContext. Let’s consider this application

**new** SpringApplicationBuilder()

.bannerMode(Banner.Mode.OFF)

.sources(demo.MyApp.**class**)

.run(args);

used with the following configuration:

**spring.main.sources**=com.acme.Config,com.acme.ExtraConfig **spring.main.banner-mode**=console

The actual application will *now* show the banner (as overridden by configuration) and use three sources for the ApplicationContext (in that order): demo.MyApp, com.acme.Config, com.acme.ExtraConfig.

## Change the location of external properties of an application

By default properties from different sources are added to the Spring Environment in a defined order (see [*Chapter 24, Externalized Configuration*](#_bookmark97) in the ‘Spring Boot features’ section for the exact order).

A nice way to augment and modify this is to add @PropertySource annotations to your application sources. Classes passed to the SpringApplication static convenience methods, and those added using setSources() are inspected to see if they have @PropertySources, and if they do, those properties are added to the Environment early enough to be used in all phases of the ApplicationContext lifecycle. Properties added in this way have lower priority than any added using the default locations (e.g. application.properties), system properties, environment variables or the command line.

You can also provide System properties (or environment variables) to change the behavior:

* spring.config.name (SPRING\_CONFIG\_NAME), defaults to application as the root of the file name.
* spring.config.location (SPRING\_CONFIG\_LOCATION) is the file to load (e.g. a classpath resource or a URL). A separate Environment property source is set up for this document and it can be overridden by system properties, environment variables or the command line.

No matter what you set in the environment, Spring Boot will always load application.properties as described above. If YAML is used then files with the ‘.yml’ extension are also added to the list by default.

Spring Boot logs the configuration files that are loaded at DEBUG level and the candidates it has not found at TRACE level.

See [ConfigFileApplicationListener](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/config/ConfigFileApplicationListener.java) for more detail.

## Use ‘short’ command line arguments

Some people like to use (for example) --port=9000 instead of --server.port=9000 to set configuration properties on the command line. You can easily enable this by using placeholders in application.properties, e.g.

**server.port**=${port:8080}

**Tip**

If you are inheriting from the spring-boot-starter-parent POM, the default filter token of the maven-resources-plugins has been changed from ${\*} to @ (i.e. @maven.token@ instead of ${maven.token}) to prevent conflicts with Spring-style placeholders. If you have enabled maven filtering for the application.properties directly, you may want to also change the default filter token to use [other delimiters](http://maven.apache.org/plugins/maven-resources-plugin/resources-mojo.html#delimiters).

**Note**

In this specific case the port binding will work in a PaaS environment like Heroku and Cloud Foundry, since in those two platforms the PORT environment variable is set automatically and Spring can bind to capitalized synonyms for Environment properties.

## Use YAML for external properties

YAML is a superset of JSON and as such is a very convenient syntax for storing external properties in a hierarchical format. E.g.

**spring**:

**application**:

**name**: cruncher

**datasource**:

**driverClassName**: com.mysql.jdbc.Driver

**url**: jdbc:mysql://localhost/test

**server**:

**port**: 9000

Create a file called application.yml and stick it in the root of your classpath, and also add snakeyaml to your dependencies (Maven coordinates org.yaml:snakeyaml, already included if you use the spring-boot-starter). A YAML file is parsed to a Java Map<String,Object> (like a JSON object), and Spring Boot flattens the map so that it is 1-level deep and has period-separated keys, a lot like people are used to with Properties files in Java.

The example YAML above corresponds to an application.properties file

**spring.application.name**=cruncher **spring.datasource.driverClassName**=com.mysql.jdbc.Driver **spring.datasource.url**=jdbc:mysql://localhost/test **server.port**=9000

See [*Section 24.6, “Using YAML instead of Properties”*](#_bookmark103) in the ‘Spring Boot features’ section for more information about YAML.

## Set the active Spring profiles

The Spring Environment has an API for this, but normally you would set a System property (spring.profiles.active) or an OS environment variable (SPRING\_PROFILES\_ACTIVE). E.g. launch your application with a -D argument (remember to put it before the main class or jar archive):

$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar

In Spring Boot you can also set the active profile in application.properties, e.g.

**spring.profiles.active**=production

A value set this way is replaced by the System property or environment variable setting, but not by the SpringApplicationBuilder.profiles() method. Thus the latter Java API can be used to augment the profiles without changing the defaults.

See [*Chapter 25, Profiles*](#_bookmark115) in the ‘Spring Boot features’ section for more information.

## Change configuration depending on the environment

A YAML file is actually a sequence of documents separated by --- lines, and each document is parsed separately to a flattened map.

If a YAML document contains a spring.profiles key, then the profiles value (comma-separated list of profiles) is fed into the Spring Environment.acceptsProfiles() and if any of those profiles is active that document is included in the final merge (otherwise not).

Example:

**server**:

**port**: 9000

*---*

**spring**:

**profiles**: development

**server**:

**port**: 9001

*---*

**spring**:

**profiles**: production

**server**:

**port**: 0

In this example the default port is 9000, but if the Spring profile ‘development’ is active then the port is 9001, and if ‘production’ is active then it is 0.

The YAML documents are merged in the order they are encountered (so later values override earlier ones).

To do the same thing with properties files you can use application-${profile}.properties to specify profile-specific values.

## Discover built-in options for external properties

Spring Boot binds external properties from application.properties (or .yml) (and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a configprops endpoint that shows all the bound and bindable properties available through @ConfigurationProperties.

The appendix includes an [application.properties](#_bookmark569) example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for @ConfigurationProperties and @Value annotations, as well as the occasional use of RelaxedPropertyResolver.

# Embedded servlet containers

## Add a Servlet, Filter or Listener to an application

There are two ways to add Servlet, Filter, ServletContextListener and the other listeners supported by the Servlet spec to your application. You can either provide Spring beans for them, or enable scanning for Servlet components.

### Add a Servlet, Filter or Listener using a Spring bean

To add a Servlet, Filter, or Servlet \*Listener provide a @Bean definition for it. This can be very useful when you want to inject configuration or dependencies. However, you must be very careful that they don’t cause eager initialization of too many other beans because they have to be installed in the container very early in the application lifecycle (e.g. it’s not a good idea to have them depend on your DataSource or JPA configuration). You can work around restrictions like that by initializing them lazily when first used instead of on initialization.

In the case of Filters and Servlets you can also add mappings and init parameters by adding a FilterRegistrationBean or ServletRegistrationBean instead of or as well as the underlying component.

**Note**

If no dispatcherType is specified on a filter registration, it will match FORWARD,INCLUDE and

REQUEST. If async has been enabled, it will match ASYNC as well.

If you are migrating a filter that has no dispatcher element in web.xml you will need to specify a dispatcherType yourself:

@Bean

**public** FilterRegistrationBean myFilterRegistration() { FilterRegistrationBean registration = **new** FilterRegistrationBean(); registration.setDispatcherTypes(DispatcherType.REQUEST);

....

**return** registration;

}

#### Disable registration of a Servlet or Filter

As [described above](#_bookmark458) any Servlet or Filter beans will be registered with the servlet container automatically. To disable registration of a particular Filter or Servlet bean create a registration bean for it and mark it as disabled. For example:

@Bean

**public** FilterRegistrationBean registration(MyFilter filter) { FilterRegistrationBean registration = **new** FilterRegistrationBean(filter); registration.setEnabled(false);

**return** registration;

}

### Add Servlets, Filters, and Listeners using classpath scanning

@WebServlet, @WebFilter, and @WebListener annotated classes can be automatically registered with an embedded servlet container by annotating a @Configuration class with

@ServletComponentScan and specifying the package(s) containing the components that you want to register. By default, @ServletComponentScan will scan from the package of the annotated class.

## Change the HTTP port

In a standalone application the main HTTP port defaults to 8080, but can be set with server.port (e.g. in application.properties or as a System property). Thanks to relaxed binding of Environment values you can also use SERVER\_PORT (e.g. as an OS environment variable).

To switch off the HTTP endpoints completely, but still create a WebApplicationContext, use

server.port=-1 (this is sometimes useful for testing).

For more details look at [*the section called “Customizing embedded servlet containers”*](#_bookmark152) in the ‘Spring Boot features’ section, or the [ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java) source code.

## Use a random unassigned HTTP port

To scan for a free port (using OS natives to prevent clashes) use server.port=0.

## Discover the HTTP port at runtime

You can access the port the server is running on from log output or from the EmbeddedWebApplicationContext via its EmbeddedServletContainer. The best way to get that and be sure that it has initialized is to add a @Bean of type ApplicationListener<EmbeddedServletContainerInitializedEvent> and pull the container out of the event when it is published.

Tests that use @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT) can also inject the actual port into a field using the @LocalServerPort annotation. For example:

@RunWith(SpringJUnit4ClassRunner.class) @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT) **public class** MyWebIntegrationTests {

@Autowired EmbeddedWebApplicationContext server;

@LocalServerPort

**int** port;

*// ...*

}

**Note**

@LocalServerPort is a meta-annotation for @Value("${local.server.port}"). Don’t try to inject the port in a regular application. As we just saw, the value is only set once the container has initialized; contrary to a test, application code callbacks are processed early (i.e. before the value is actually available).

## Configure SSL

SSL can be configured declaratively by setting the various server.ssl.\* properties, typically in

application.properties or application.yml. For example:

**server.port**=8443

**server.ssl.key-store**=classpath:keystore.jks **server.ssl.key-store-password**=secret **server.ssl.key-password**=another-secret

See [Ssl](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/embedded/Ssl.java) for details of all of the supported properties.

Using configuration like the example above means the application will no longer support plain HTTP connector at port 8080. Spring Boot doesn’t support the configuration of both an HTTP connector and an HTTPS connector via application.properties. If you want to have both then you’ll need to configure one of them programmatically. It’s recommended to use application.properties to configure HTTPS as the HTTP connector is the easier of the two to configure programmatically. See the [spring-boot-sample-tomcat-multi-connectors](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-tomcat-multi-connectors) sample project for an example.

## Configure Access Logging

Access logs can be configured for Tomcat and Undertow via their respective namespaces. For instance, the following logs access on Tomcat with a [custom pattern](https://tomcat.apache.org/tomcat-8.0-doc/config/valve.html#Access_Logging).

**server.tomcat.basedir**=my-tomcat **server.tomcat.accesslog.enabled**=true **server.tomcat.accesslog.pattern**=%t %a "%r" %s (%D ms)

**Note**

The default location for logs is a logs directory relative to the tomcat base dir and said directory is a temp directory by default so you may want to fix Tomcat’s base directory or use an absolute path for the logs. In the example above, the logs will be available in my-tomcat/logs relative to the working directory of the application.

Access logging for undertow can be configured in a similar fashion

**server.undertow.accesslog.enabled**=true **server.undertow.accesslog.pattern**=%t %a "%r" %s (%D ms)

Logs are stored in a logs directory relative to the working directory of the application. This can be customized via server.undertow.accesslog.directory.

## Use behind a front-end proxy server

Your application might need to send 302 redirects or render content with absolute links back to itself. When running behind a proxy, the caller wants a link to the proxy, and not to the physical address of the machine hosting your app. Typically such situations are handled via a contract with the proxy, which will add headers to tell the back end how to construct links to itself.

If the proxy adds conventional X-Forwarded-For and X-Forwarded-Proto headers (most do this out of the box) the absolute links should be rendered correctly as long as server.use-forward- headers is set to true in your application.properties.

**Note**

If your application is running in Cloud Foundry or Heroku the server.use-forward-headers

property will default to true if not specified. In all other instances it defaults to false.

### Customize Tomcat’s proxy configuration

If you are using Tomcat you can additionally configure the names of the headers used to carry “forwarded” information:

server.tomcat.remote-ip-header=x-your-remote-ip-header server.tomcat.protocol-header=x-your-protocol-header

Tomcat is also configured with a default regular expression that matches internal proxies that are to be trusted. By default, IP addresses in 10/8, 192.168/16, 169.254/16 and 127/8 are trusted. You can customize the valve’s configuration by adding an entry to application.properties, e.g.

server.tomcat.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}

**Note**

The double backslashes are only required when you’re using a properties file for configuration. If you are using YAML, single backslashes are sufficient and a value that’s equivalent to the one shown above would be 192\.168\.\d{1,3}\.\d{1,3}.

**Note**

You can trust all proxies by setting the internal-proxies to empty (but don’t do this in production).

You can take complete control of the configuration of Tomcat’s RemoteIpValve by switching the automatic one off (i.e. set server.use-forward-headers=false) and adding a new valve instance in a TomcatEmbeddedServletContainerFactory bean.

## Configure Tomcat

Generally you can follow the advice from [*Section 72.8, “Discover built-in options for external properties”*](#_bookmark455)about @ConfigurationProperties (ServerProperties is the main one here), but also look at EmbeddedServletContainerCustomizer and various Tomcat-specific \*Customizers that you can add in one of those. The Tomcat APIs are quite rich so once you have access to the TomcatEmbeddedServletContainerFactory you can modify it in a number of ways. Or the nuclear option is to add your own TomcatEmbeddedServletContainerFactory.

## Enable Multiple Connectors with Tomcat

Add a org.apache.catalina.connector.Connector to the TomcatEmbeddedServletContainerFactory which can allow multiple connectors, e.g. HTTP and HTTPS connector:

@Bean

**public** EmbeddedServletContainerFactory servletContainer() { TomcatEmbeddedServletContainerFactory tomcat = **new** TomcatEmbeddedServletContainerFactory(); tomcat.addAdditionalTomcatConnectors(createSslConnector());

**return** tomcat;

}

**private** Connector createSslConnector() {

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

**try** {

File keystore = **new** ClassPathResource(***"keystore"***).getFile(); File truststore = **new** ClassPathResource(***"keystore"***).getFile(); connector.setScheme(***"https"***);

connector.setSecure(true); connector.setPort(8443); protocol.setSSLEnabled(true);

protocol.setKeystoreFile(keystore.getAbsolutePath()); protocol.setKeystorePass(***"changeit"***); protocol.setTruststoreFile(truststore.getAbsolutePath()); protocol.setTruststorePass(***"changeit"***); protocol.setKeyAlias(***"apitester"***);

**return** connector;

}

**catch** (IOException ex) {

**throw new** IllegalStateException(***"can't access keystore: ["*** + ***"keystore"***

+ ***"] or truststore: ["*** + ***"keystore"*** + ***"]"***, ex);

}

}

## Use Tomcat’s LegacyCookieProcessor

The embedded Tomcat used by Spring Boot does not support "Version 0" of the Cookie format out of the box, and you may see the following error:

java.lang.IllegalArgumentException: An invalid character [32] was present in the Cookie value

If at all possible, you should consider updating your code to only store values compliant with later Cookie specifications. If, however, you’re unable to change the way that cookies are written, you can instead configure Tomcat to use a LegacyCookieProcessor. To switch to the LegacyCookieProcessor use an EmbeddedServletContainerCustomizer bean that adds a TomcatContextCustomizer:

@Bean

**public** EmbeddedServletContainerCustomizer cookieProcessorCustomizer() {

**return new** EmbeddedServletContainerCustomizer() {

@Override

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

**if** (container **instanceof** TomcatEmbeddedServletContainerFactory) { ((TomcatEmbeddedServletContainerFactory) container)

.addContextCustomizers(**new** TomcatContextCustomizer() {

@Override

**public void** customize(Context context) { context.setCookieProcessor(**new** LegacyCookieProcessor());

}

});

}

}

};

}

## Use Jetty instead of Tomcat

The Spring Boot starters (spring-boot-starter-web in particular) use Tomcat as an embedded container by default. You need to exclude those dependencies and include the Jetty one instead. Spring Boot provides Tomcat and Jetty dependencies bundled together as separate starters to help make this process as easy as possible.

Example in Maven:

**<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>**

Example in Gradle:

configurations {

compile.exclude module: ***"spring-boot-starter-tomcat"***

}

dependencies {

compile(***"org.springframework.boot:spring-boot-starter-web:1.5.8.RELEASE"***) compile(***"org.springframework.boot:spring-boot-starter-jetty:1.5.8.RELEASE"***)

*// ...*

}

## Configure Jetty

Generally you can follow the advice from [*Section 72.8, “Discover built-in options for external properties”*](#_bookmark455)about @ConfigurationProperties (ServerProperties is the main one here), but also look at EmbeddedServletContainerCustomizer. The Jetty APIs are quite rich so once you have access to the JettyEmbeddedServletContainerFactory you can modify it in a number of ways. Or the nuclear option is to add your own JettyEmbeddedServletContainerFactory.

## Use Undertow instead of Tomcat

Using Undertow instead of Tomcat is very similar to [using Jetty instead of Tomcat](#_bookmark471). You need to exclude the Tomcat dependencies and include the Undertow starter instead.

Example in Maven:

**<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-undertow**</artifactId>**

**</dependency>**

Example in Gradle:

configurations {

compile.exclude module: ***"spring-boot-starter-tomcat"***

}

dependencies {

compile(***"org.springframework.boot:spring-boot-starter-web:1.5.8.RELEASE"***) compile(***"org.springframework.boot:spring-boot-starter-undertow:1.5.8.RELEASE"***)

*// ...*

}

## Configure Undertow

Generally you can follow the advice from [*Section 72.8, “Discover built-in options for external properties”*](#_bookmark455)about @ConfigurationProperties (ServerProperties and ServerProperties.Undertow are the main ones here), but also look at EmbeddedServletContainerCustomizer. Once you have access to the UndertowEmbeddedServletContainerFactory you can use an UndertowBuilderCustomizer to modify Undertow’s configuration to meet your needs. Or the nuclear option is to add your own UndertowEmbeddedServletContainerFactory.

## Enable Multiple Listeners with Undertow

Add an UndertowBuilderCustomizer to the UndertowEmbeddedServletContainerFactory

and add a listener to the Builder:

@Bean

**public** UndertowEmbeddedServletContainerFactory embeddedServletContainerFactory() { UndertowEmbeddedServletContainerFactory factory = **new** UndertowEmbeddedServletContainerFactory(); factory.addBuilderCustomizers(**new** UndertowBuilderCustomizer() {

@Override

**public void** customize(Builder builder) { builder.addHttpListener(8080, ***"0.0.0.0"***);

}

});

**return** factory;

}

## Use Tomcat 7.x or 8.0

Tomcat 7 & 8.0 work with Spring Boot, but the default is to use Tomcat 8.5. If you cannot use Tomcat

8.5 (for example, because you are using Java 1.6) you will need to change your classpath to reference a different version.

### Use Tomcat 7.x or 8.0 with Maven

If you are using the starters and parent you can change the Tomcat version property and additionally import tomcat-juli. E.g. for a simple webapp or service:

**<properties>**

**<tomcat.version>**7.0.59**</tomcat.version>**

**</properties>**

**<dependencies>**

...

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>**tomcat-juli**</artifactId>**

**<version>**${tomcat.version}**</version>**

**</dependency>**

...

**</dependencies>**

### Use Tomcat 7.x or 8.0 with Gradle

With Gradle, you can change the Tomcat version by setting the tomcat.version property and then additionally include tomcat-juli:

ext[***'tomcat.version'***] = ***'7.0.59'***

dependencies {

compile ***'org.springframework.boot:spring-boot-starter-web'***

compile group:***'org.apache.tomcat'***, name:***'tomcat-juli'***, version:property(***'tomcat.version'***)

}

## Use Jetty 9.2

Jetty 9.2 works with Spring Boot, but the default is to use Jetty 9.3. If you cannot use Jetty 9.3 (for example, because you are using Java 7) you will need to change your classpath to reference Jetty 9.2.

### Use Jetty 9.2 with Maven

If you are using the starters and parent you can just add the Jetty starter and override the

jetty.version property:

**<properties>**

**<jetty.version>**9.2.17.v20160517**</jetty.version>**

**</properties>**

**<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>**

**</dependencies>**

### Use Jetty 9.2 with Gradle

You can set the jetty.version property. For example, for a simple webapp or service:

ext[***'jetty.version'***] = ***'9.2.17.v20160517'***

dependencies {

compile (***'org.springframework.boot:spring-boot-starter-web'***) {

exclude group: ***'org.springframework.boot'***, module: ***'spring-boot-starter-tomcat'***

}

compile (***'org.springframework.boot:spring-boot-starter-jetty'***)

}

## Use Jetty 8

Jetty 8 works with Spring Boot, but the default is to use Jetty 9.3. If you cannot use Jetty 9.3 (for example, because you are using Java 1.6) you will need to change your classpath to reference Jetty 8. You will also need to exclude Jetty’s WebSocket-related dependencies.

### Use Jetty 8 with Maven

If you are using the starters and parent you can just add the Jetty starter with the required WebSocket exclusion and change the version properties, e.g. for a simple webapp or service:

**<properties>**

**<jetty.version>**8.1.15.v20140411**</jetty.version>**

**<jetty-jsp.version>**2.2.0.v201112011158**</jetty-jsp.version>**

**</properties>**

**<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>**

**<exclusions>**

**<exclusion>**

**<groupId>**org.eclipse.jetty.websocket**</groupId>**

**<artifactId>**\***</artifactId>**

**</exclusion>**

**</exclusions>**

**</dependency>**

**</dependencies>**

### Use Jetty 8 with Gradle

You can set the jetty.version property and exclude the WebSocket dependency, e.g. for a simple webapp or service:

ext[***'jetty.version'***] = ***'8.1.15.v20140411'***

dependencies {

compile (***'org.springframework.boot:spring-boot-starter-web'***) {

exclude group: ***'org.springframework.boot'***, module: ***'spring-boot-starter-tomcat'***

}

compile (***'org.springframework.boot:spring-boot-starter-jetty'***) { exclude group: ***'org.eclipse.jetty.websocket'***

}

}

## Create WebSocket endpoints using @ServerEndpoint

If you want to use @ServerEndpoint in a Spring Boot application that used an embedded container, you must declare a single ServerEndpointExporter @Bean:

@Bean

**public** ServerEndpointExporter serverEndpointExporter() {

**return new** ServerEndpointExporter();

}

This bean will register any @ServerEndpoint annotated beans with the underlying WebSocket container. When deployed to a standalone servlet container this role is performed by a servlet container initializer and the ServerEndpointExporter bean is not required.

## Enable HTTP response compression

HTTP response compression is supported by Jetty, Tomcat, and Undertow. It can be enabled via

application.properties:

**server.compression.enabled**=true

By default, responses must be at least 2048 bytes in length for compression to be performed. This can be configured using the server.compression.min-response-size property.

By default, responses will only be compressed if their content type is one of the following:

* text/html
* text/xml
* text/plain
* text/css

This can be configured using the server.compression.mime-types property.

# Spring MVC

## Write a JSON REST service

Any Spring @RestController in a Spring Boot application should render JSON response by default as long as Jackson2 is on the classpath. For example:

@RestController

**public class** MyController {

@RequestMapping("/thing")

**public** MyThing thing() {

**return new** MyThing();

}

}

As long as MyThing can be serialized by Jackson2 (e.g. a normal POJO or Groovy object) then localhost:8080/thing will serve a JSON representation of it by default. Sometimes in a browser you might see XML responses because browsers tend to send accept headers that prefer XML.

## Write an XML REST service

If you have the Jackson XML extension (jackson-dataformat-xml) on the classpath, it will be used to render XML responses and the very same example as we used for JSON would work. To use it, add the following dependency to your project:

**<dependency>**

**<groupId>**com.fasterxml.jackson.dataformat**</groupId>**

**<artifactId>**jackson-dataformat-xml**</artifactId>**

**</dependency>**

You may also want to add a dependency on Woodstox. It’s faster than the default StAX implementation provided by the JDK and also adds pretty print support and improved namespace handling:

**<dependency>**

**<groupId>**org.codehaus.woodstox**</groupId>**

**<artifactId>**woodstox-core-asl**</artifactId>**

**</dependency>**

If Jackson’s XML extension is not available, JAXB (provided by default in the JDK) will be used, with the additional requirement to have MyThing annotated as @XmlRootElement:

@XmlRootElement

**public class** MyThing {

**private** String name;

*// .. getters and setters*

}

To get the server to render XML instead of JSON you might have to send an Accept: text/xml

header (or use a browser).

## Customize the Jackson ObjectMapper

Spring MVC (client and server side) uses HttpMessageConverters to negotiate content conversion in an HTTP exchange. If Jackson is on the classpath you already get the default converter(s) provided by Jackson2ObjectMapperBuilder, an instance of which is auto-configured for you.

The ObjectMapper (or XmlMapper for Jackson XML converter) instance created by default has the following customized properties:

* MapperFeature.DEFAULT\_VIEW\_INCLUSION is disabled
* DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES is disabled Spring Boot has also some features to make it easier to customize this behavior.

You can configure the ObjectMapper and XmlMapper instances using the environment. Jackson provides an extensive suite of simple on/off features that can be used to configure various aspects of its processing. These features are described in six enums in Jackson which map onto properties in the environment:

|  |  |
| --- | --- |
| **Jackson enum** | **Environment property** |
| com.fasterxml.jackson.databind.Deseriaslpirziantgi.ojnaFcekastounr.edeserialization.<featu | |
| false | |
| com.fasterxml.jackson.core.JsonGeneratsoprr.iFnega.tjuarcekson.generator.<feature\_nam | |
| false | |
| com.fasterxml.jackson.databind.MapperFsepartiunrge.jackson.mapper.<feature\_name>=  false | |
| com.fasterxml.jackson.core.JsonParser.sFperaitnugr.ejackson.parser.<feature\_name>=  false | |
| com.fasterxml.jackson.databind.Serialiszpartiinogn.Fjeaactkusroen.serialization.<feature | |
| false | |
| com.fasterxml.jackson.annotation.JsonIsnpcrliundge..jIanccklsuodne.default-property- | |
| inclusion=always|non\_null| | |
| non\_absent|non\_default|non\_empty | |

re\_name>=tru

e>=true|

true|

true|

\_name>=true|

For example, to enable pretty print, set spring.jackson.serialization.indent\_output=true. Note that, thanks to the use of [relaxed](#_bookmark111) [binding](#_bookmark111), the case of indent\_output doesn’t have to match the case of the corresponding enum constant which is INDENT\_OUTPUT.

This environment-based configuration is applied to the auto-configured Jackson2ObjectMapperBuilder bean, and will apply to any mappers created using the builder, including the auto-configured ObjectMapper bean.

The context’s Jackson2ObjectMapperBuilder can be customized by one or more Jackson2ObjectMapperBuilderCustomizer beans. Such customizer beans can be ordered and Boot’s own customizer has an order of 0, allowing additional customization to be applied both before and after Boot’s customization.

Any beans of type com.fasterxml.jackson.databind.Module will be automatically registered with the auto-configured Jackson2ObjectMapperBuilder and applied to any ObjectMapper instances that it creates. This provides a global mechanism for contributing custom modules when you add new features to your application.

If you want to replace the default ObjectMapper completely, either define a @Bean of that type and mark it as @Primary, or, if you prefer the builder-based approach, define a

Jackson2ObjectMapperBuilder @Bean. Note that in either case this will disable all auto- configuration of the ObjectMapper.

If you provide any @Beans of type MappingJackson2HttpMessageConverter then they will replace the default value in the MVC configuration. Also, a convenience bean is provided of type HttpMessageConverters (always available if you use the default MVC configuration) which has some useful methods to access the default and user-enhanced message converters.

See also the [*Section 74.4, “Customize the @ResponseBody rendering”*](#_bookmark491)section and the

[WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java) source code for more details.

## Customize the @ResponseBody rendering

Spring uses HttpMessageConverters to render @ResponseBody (or responses from @RestController). You can contribute additional converters by simply adding beans of that type in a Spring Boot context. If a bean you add is of a type that would have been included by default anyway (like MappingJackson2HttpMessageConverter for JSON conversions) then it will replace the default value. A convenience bean is provided of type HttpMessageConverters (always available if you use the default MVC configuration) which has some useful methods to access the default and user- enhanced message converters (useful, for example if you want to manually inject them into a custom RestTemplate).

As in normal MVC usage, any WebMvcConfigurerAdapter beans that you provide can also contribute converters by overriding the configureMessageConverters method, but unlike with normal MVC, you can supply only additional converters that you need (because Spring Boot uses the same mechanism to contribute its defaults). Finally, if you opt-out of the Spring Boot default MVC configuration by providing your own @EnableWebMvc configuration, then you can take control completely and do everything manually using getMessageConverters from WebMvcConfigurationSupport.

See the [WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java) source code for more details.

## Handling Multipart File Uploads

Spring Boot embraces the Servlet 3 javax.servlet.http.Part API to support uploading files. By default Spring Boot configures Spring MVC with a maximum file of 1MB per file and a maximum of 10MB of file data in a single request. You may override these values, as well as the location to which intermediate data is stored (e.g., to the /tmp directory) and the threshold past which data is flushed to disk by using the properties exposed in the MultipartProperties class. If you want to specify that files be unlimited, for example, set the spring.http.multipart.max-file-size property to -1.

The multipart support is helpful when you want to receive multipart encoded file data as a @RequestParam-annotated parameter of type MultipartFile in a Spring MVC controller handler method.

See the [MultipartAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/MultipartAutoConfiguration.java) source for more details.

## Switch off the Spring MVC DispatcherServlet

Spring Boot wants to serve all content from the root of your application / down. If you would rather map your own servlet to that URL you can do it, but of course you may lose some of the other Boot MVC features. To add your own servlet and map it to the root resource just declare a @Bean of type Servlet

and give it the special bean name dispatcherServlet (You can also create a bean of a different type with that name if you want to switch it off and not replace it).

## Switch off the Default MVC configuration

The easiest way to take complete control over MVC configuration is to provide your own @Configuration with the @EnableWebMvc annotation. This will leave all MVC configuration in your hands.

## Customize ViewResolvers

A ViewResolver is a core component of Spring MVC, translating view names in @Controller to actual View implementations. Note that ViewResolvers are mainly used in UI applications, rather than REST-style services (a View is not used to render a @ResponseBody). There are many implementations of ViewResolver to choose from, and Spring on its own is not opinionated about which ones you should use. Spring Boot, on the other hand, installs one or two for you depending on what it finds on the classpath and in the application context. The DispatcherServlet uses all the resolvers it finds in the application context, trying each one in turn until it gets a result, so if you are adding your own you have to be aware of the order and in which position your resolver is added.

WebMvcAutoConfiguration adds the following ViewResolvers to your context:

* An InternalResourceViewResolver with bean id ‘defaultViewResolver’. This one locates physical resources that can be rendered using the DefaultServlet (e.g. static resources and JSP pages if you are using those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (defaults are both empty, but accessible for external configuration via spring.mvc.view.prefix and spring.mvc.view.suffix). It can be overridden by providing a bean of the same type.
* A BeanNameViewResolver with id ‘beanNameViewResolver’. This is a useful member of the view resolver chain and will pick up any beans with the same name as the View being resolved. It shouldn’t be necessary to override or replace it.
* A ContentNegotiatingViewResolver with id ‘viewResolver’ is only added if there **are** actually beans of type View present. This is a ‘master’ resolver, delegating to all the others and attempting to find a match to the ‘Accept’ HTTP header sent by the client. There is a useful [blog about ContentNegotiatingViewResolver](https://spring.io/blog/2013/06/03/content-negotiation-using-views) that you might like to study to learn more, and also look at the source code for detail. You can switch off the auto-configured ContentNegotiatingViewResolver by defining a bean named ‘viewResolver’.
* If you use Thymeleaf you will also have a ThymeleafViewResolver with id ‘thymeleafViewResolver’. It looks for resources by surrounding the view name with a prefix and suffix (externalized to spring.thymeleaf.prefix and spring.thymeleaf.suffix, defaults ‘classpath:/templates/’ and ‘.html’ respectively). It can be overridden by providing a bean of the same name.
* If you use FreeMarker you will also have a FreeMarkerViewResolver with id ‘freeMarkerViewResolver’. It looks for resources in a loader path (externalized to spring.freemarker.templateLoaderPath, default ‘classpath:/templates/’) by surrounding the view name with a prefix and suffix (externalized to spring.freemarker.prefix and spring.freemarker.suffix, with empty and ‘.ftl’ defaults respectively). It can be overridden by providing a bean of the same name.
* If you use Groovy templates (actually if groovy-templates is on your classpath) you will also have a GroovyMarkupViewResolver with id ‘groovyMarkupViewResolver’. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to spring.groovy.template.prefix and spring.groovy.template.suffix, defaults ‘classpath:/templates/’ and ‘.tpl’ respectively). It can be overridden by providing a bean of the same name.

Check out [WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java), [ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java), [FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java) and [GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java)

## Use Thymeleaf 3

By default, spring-boot-starter-thymeleaf uses Thymeleaf 2.1. If you are using the spring- boot-starter-parent, you can use Thymeleaf 3 by overriding the thymeleaf.version and thymeleaf-layout-dialect.version properties, for example:

**<properties>**

**<thymeleaf.version>**3.0.2.RELEASE**</thymeleaf.version>**

**<thymeleaf-layout-dialect.version>**2.1.1**</thymeleaf-layout-dialect.version>**

**</properties>**

**Note**

if you are managing dependencies yourself, look at spring-boot-dependencies for the list of artifacts that are related to those two versions.

To avoid a warning message about the HTML 5 template mode being deprecated and the HTML template mode being used instead, you may also want to explicitly configure spring.thymeleaf.mode to be HTML, for example:

**spring.thymeleaf.mode**: HTML

Please refer to the [Thymeleaf 3 sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-web-thymeleaf3) to see this in action.

If you are using any of the other auto-configured Thymeleaf Extras (Spring Security, Data Attribute, or Java 8 Time) you should also override each of their versions to one that is compatible with Thymeleaf 3.0.

# HTTP clients

## Configure RestTemplate to use a proxy

As described in [Section 33.1, “RestTemplate customization”](#_bookmark244), a RestTemplateCustomizer can be used with RestTemplateBuilder to build a customized RestTemplate. This is the recommended approach for creating a RestTemplate configured to use a proxy.

The exact details of the proxy configuration depend on the underlying client request factory that is being used. Here’s an example of configuring HttpComponentsClientRequestFactory with an HttpClient that uses a proxy for all hosts except 192.168.0.5.

**static class** ProxyCustomizer **implements** RestTemplateCustomizer {

@Override

**public void** customize(RestTemplate restTemplate) { HttpHost proxy = **new** HttpHost(***"proxy.example.com"***); HttpClient httpClient = HttpClientBuilder.create()

.setRoutePlanner(**new** DefaultProxyRoutePlanner(proxy) {

@Override

**public** HttpHost determineProxy(HttpHost target, HttpRequest request, HttpContext context)

**throws** HttpException {

**if** (target.getHostName().equals(***"192.168.0.5"***)) {

**return** null;

}

**return super**.determineProxy(target, request, context);

}

}).build(); restTemplate.setRequestFactory(

**new** HttpComponentsClientHttpRequestFactory(httpClient));

}

}

# Logging

Spring Boot has no mandatory logging dependency, except for the Commons Logging API, of which there are many implementations to choose from. To use [Logback](http://logback.qos.ch/) you need to include it and jcl-over- slf4j (which implements the Commons Logging API) on the classpath. The simplest way to do that is through the starters which all depend on spring-boot-starter-logging. For a web application you only need spring-boot-starter-web since it depends transitively on the logging starter. For example, using Maven:

**<dependency>**

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

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

**</dependency>**

Spring Boot has a LoggingSystem abstraction that attempts to configure logging based on the content of the classpath. If Logback is available it is the first choice.

If the only change you need to make to logging is to set the levels of various loggers then you can do that in application.properties using the "logging.level" prefix, e.g.

**logging.level.org.springframework.web**=DEBUG **logging.level.org.hibernate**=ERROR

You can also set the location of a file to log to (in addition to the console) using "logging.file".

To configure the more fine-grained settings of a logging system you need to use the native configuration format supported by the LoggingSystem in question. By default Spring Boot picks up the native configuration from its default location for the system (e.g. classpath:logback.xml for Logback), but you can set the location of the config file using the "logging.config" property.

## Configure Logback for logging

If you put a logback.xml in the root of your classpath it will be picked up from there (or logback- spring.xml to take advantage of the templating features provided by Boot). Spring Boot provides a default base configuration that you can include if you just want to set levels, e.g.

<?xml version="1.0" encoding="UTF-8"?>

**<configuration>**

**<include resource**=**"org/springframework/boot/logging/logback/base.xml"/>**

**<logger name**=**"org.springframework.web" level**=**"DEBUG"/>**

**</configuration>**

If you look at that base.xml in the spring-boot jar, you will see that it uses some useful System properties which the LoggingSystem takes care of creating for you. These are:

* ${PID} the current process ID.
* ${LOG\_FILE} if logging.file was set in Boot’s external configuration.
* ${LOG\_PATH} if logging.path was set (representing a directory for log files to live in).
* ${LOG\_EXCEPTION\_CONVERSION\_WORD} if logging.exception-conversion-word was set in Boot’s external configuration.

Spring Boot also provides some nice ANSI colour terminal output on a console (but not in a log file) using a custom Logback converter. See the default base.xml configuration for details.

If Groovy is on the classpath you should be able to configure Logback with logback.groovy as well (it will be given preference if present).

### Configure logback for file only output

If you want to disable console logging and write output only to a file you need a custom logback- spring.xml that imports file-appender.xml but not console-appender.xml:

<?xml version="1.0" encoding="UTF-8"?>

**<configuration>**

**<include resource**=**"org/springframework/boot/logging/logback/defaults.xml" />**

**<property name**=**"LOG\_FILE" value**=**"${LOG\_FILE:-${LOG\_PATH:-${LOG\_TEMP:-${java.io.tmpdir:-/ tmp}}/}spring.log}"/>**

**<include resource**=**"org/springframework/boot/logging/logback/file-appender.xml" />**

**<root level**=**"INFO">**

**<appender-ref ref**=**"FILE" />**

**</root>**

**</configuration>**

You also need to add logging.file to your application.properties:

**logging.file**=myapplication.log

## Configure Log4j for logging

Spring Boot supports [Log4j 2](http://logging.apache.org/log4j/2.x) 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. If you aren’t using the starters then you need to provide jcl-over-slf4j (at least) in addition to Log4j 2.

The simplest path is probably through the starters, even though it requires some jiggling with excludes, .e.g. in Maven:

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

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

**<exclusions>**

**<exclusion>**

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

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

**</exclusion>**

**</exclusions>**

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-starter-log4j2**</artifactId>**

**</dependency>**

**Note**

The use of the Log4j starters gathers together the dependencies for common logging requirements (e.g. including having Tomcat use java.util.logging but configuring the output using Log4j 2). See the Actuator Log4j 2 samples for more detail and to see it in action.

### Use YAML or JSON to configure Log4j 2

In addition to its default XML configuration format, Log4j 2 also supports YAML and JSON configuration files. To configure Log4j 2 to use an alternative configuration file format, add the appropriate dependencies to the classpath and name your configuration files to match your chosen file format:

|  |  |  |
| --- | --- | --- |
| **Format** | **Dependencies** | **File names** |
| YAML | com.fasterxml.jackson.core:jackson-databind | log4j2.yam |
|  | com.fasterxml.jackson.dataformat:jackson-dataformat- | log4j2.yml |
|  | yaml |  |
| JSON | com.fasterxml.jackson.core:jackson-databind | log4j2.jso |
|  |  | log4j2.jsn |

l

n

# Data Access

## Configure a custom DataSource

To configure your own DataSource define a @Bean of that type in your configuration. Spring Boot will reuse your DataSource anywhere one is required, including database initialization. If you need to externalize some settings, you can easily bind your DataSource to the environment (see [the section](#_bookmark110) [called “Third-party configuration”](#_bookmark110)).

@Bean @ConfigurationProperties(prefix="app.datasource") **public** DataSource dataSource() {

**return new** FancyDataSource();

}

**app.datasource.url**=jdbc:h2:mem:mydb **app.datasource.username**=sa **app.datasource.pool-size**=30

Assuming that your FancyDataSource has regular JavaBean properties for the url, the username and the pool size, these settings will be bound automatically before the DataSource is made available to other components. The regular [database initialization](#_bookmark520) will also happen (so the relevant sub-set of spring.datasource.\* can still be used with your custom configuration).

You can apply the same principle if you are configuring a custom JNDI DataSource:

@Bean(destroyMethod="") @ConfigurationProperties(prefix="app.datasource") **public** DataSource dataSource() **throws** Exception {

JndiDataSourceLookup dataSourceLookup = **new** JndiDataSourceLookup();

**return** dataSourceLookup.getDataSource(***"java:comp/env/jdbc/YourDS"***);

}

Spring Boot also provides a utility builder class DataSourceBuilder that can be used to create one of the standard data sources (if it is on the classpath). The builder can detect the one to use based on what’s available on the classpath. It also auto detects the driver based on the JDBC url.

@Bean @ConfigurationProperties("app.datasource") **public** DataSource dataSource() {

**return** DataSourceBuilder.create().build();

}

To run an app with that DataSource, all that is needed really is the connection information; pool- specific settings can also be provided, check the implementation that is going to be used at runtime for more details.

**app.datasource.url**=jdbc:mysql://localhost/test **app.datasource.username**=dbuser **app.datasource.password**=dbpass **app.datasource.pool-size**=30

There is a catch however. Because the actual type of the connection pool is not exposed, no keys are generated in the metadata for your custom DataSource and no completion is available in your IDE (The DataSource interface doesn’t expose any property). Also, if you happen to *only* have Hikari on the classpath, this basic setup will not work because Hikari has no url parameter (but a jdbcUrl parameter). You will have to rewrite your configuration as follows:

**app.datasource.jdbc-url**=jdbc:mysql://localhost/test **app.datasource.username**=dbuser **app.datasource.password**=dbpass **app.datasource.maximum-pool-size**=30

You can fix that by forcing the connection pool to use and return a dedicated implementation rather than DataSource. You won’t be able to change the implementation at runtime but the list of options will be explicit.

@Bean @ConfigurationProperties("app.datasource") **public** HikariDataSource dataSource() {

**return** (HikariDataSource) DataSourceBuilder.create()

.type(HikariDataSource.**class**).build();

}

You can even go further by leveraging what DataSourceProperties does for you, that is providing a default embedded database if no url is provided with a sensible username and password for it. You can easily initialize a DataSourceBuilder from the state of any DataSourceProperties so you could just as well inject the one Spring Boot creates automatically. However, that would split your configuration in two namespaces: url, username, password, type and driver on spring.datasource and the rest on your custom namespace (app.datasource). To avoid that, you can redefine a custom DataSourceProperties on your custom namespace:

@Bean @Primary

@ConfigurationProperties("app.datasource")

**public** DataSourceProperties dataSourceProperties() {

**return new** DataSourceProperties();

}

@Bean @ConfigurationProperties("app.datasource")

**public** HikariDataSource dataSource(DataSourceProperties properties) {

**return** (HikariDataSource) properties.initializeDataSourceBuilder()

.type(HikariDataSource.**class**).build();

}

This setup puts you *in pair* with what Spring Boot does for you by default, except that a dedicated connection pool is chosen (in code) and its settings are exposed in the same namespace. Because DataSourceProperties is taking care of the url/jdbcUrl translation for you, you can configure it like this:

**app.datasource.url**=jdbc:mysql://localhost/test **app.datasource.username**=dbuser **app.datasource.password**=dbpass **app.datasource.maximum-pool-size**=30

**Note**

Because your custom configuration chooses to go with Hikari, app.datasource.type will have no effect. In practice the builder will be initialized with whatever value you might set there and then overridden by the call to .type().

See [*Section 29.1, “Configure a DataSource”*](#_bookmark166)in the ‘Spring Boot features’ section and the

[DataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.java) class for more details.

## Configure Two DataSources

If you need to configure multiple data sources, you can apply the same tricks that are described in the previous section. You must, however, mark one of the DataSource @Primary as various auto- configurations down the road expect to be able to get one by type.

If you create your own DataSource, the auto-configuration will back off. In the example below, we provide the *exact* same features set than what the auto-configuration provides on the primary data source:

@Bean @Primary

@ConfigurationProperties("app.datasource.foo")

**public** DataSourceProperties fooDataSourceProperties() {

**return new** DataSourceProperties();

}

@Bean @Primary

@ConfigurationProperties("app.datasource.foo")

**public** DataSource fooDataSource() {

**return** fooDataSourceProperties().initializeDataSourceBuilder().build();

}

@Bean @ConfigurationProperties("app.datasource.bar") **public** BasicDataSource barDataSource() {

**return** (BasicDataSource) DataSourceBuilder.create()

.type(BasicDataSource.**class**).build();

}

**Tip**

fooDataSourceProperties has to be flagged @Primary so that the database initializer feature uses your copy (should you use that).

Both data sources are also bound for advanced customizations. For instance you could configure them as follows:

**app.datasource.foo.type**=com.zaxxer.hikari.HikariDataSource **app.datasource.foo.maximum-pool-size**=30

**app.datasource.bar.url**=jdbc:mysql://localhost/test **app.datasource.bar.username**=dbuser **app.datasource.bar.password**=dbpass **app.datasource.bar.max-total**=30

Of course, you can apply the same concept to the secondary DataSource as well:

@Bean @Primary

@ConfigurationProperties("app.datasource.foo")

**public** DataSourceProperties fooDataSourceProperties() {

**return new** DataSourceProperties();

}

@Bean @Primary

@ConfigurationProperties("app.datasource.foo")

**public** DataSource fooDataSource() {

**return** fooDataSourceProperties().initializeDataSourceBuilder().build();

}

@Bean @ConfigurationProperties("app.datasource.bar")

**public** DataSourceProperties barDataSourceProperties() {

**return new** DataSourceProperties();

}

@Bean @ConfigurationProperties("app.datasource.bar") **public** DataSource barDataSource() {

**return** barDataSourceProperties().initializeDataSourceBuilder().build();

}

This final example configures two data sources on custom namespaces with the same logic than what Spring Boot would do in auto-configuration.

## Use Spring Data repositories

Spring Data can create implementations for you of @Repository interfaces of various flavors. Spring Boot will handle all of that for you as long as those @Repositories are included in the same package (or a sub-package) of your @EnableAutoConfiguration class.

For many applications all you will need is to put the right Spring Data dependencies on your classpath (there is a spring-boot-starter-data-jpa for JPA and a spring-boot-starter-data- mongodb for Mongodb), create some repository interfaces to handle your @Entity objects. Examples are in the [JPA sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-data-jpa) or the [Mongodb sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-data-mongodb).

Spring Boot tries to guess the location of your @Repository definitions, based on the @EnableAutoConfiguration it finds. To get more control, use the @EnableJpaRepositories annotation (from Spring Data JPA).

## Separate @Entity definitions from Spring configuration

Spring Boot tries to guess the location of your @Entity definitions, based on the @EnableAutoConfiguration it finds. To get more control, you can use the @EntityScan annotation, e.g.

@Configuration @EnableAutoConfiguration

@EntityScan(basePackageClasses=City.class)

**public class** Application {

*//...*

}

## Configure JPA properties

Spring Data JPA already provides some vendor-independent configuration options (e.g. for SQL logging) and Spring Boot exposes those, and a few more for hibernate as external configuration properties. Some of them are automatically detected according to the context so you shouldn’t have to set them.

The spring.jpa.hibernate.ddl-auto is a special case in that it has different defaults depending on whether you are using an embedded database (create-drop) or not (none). The dialect to use is also automatically detected based on the current DataSource but you can set spring.jpa.database yourself if you want to be explicit and bypass that check on startup.

**Note**

Specifying a database leads to the configuration of a well-defined Hibernate dialect. Several databases have more than one Dialect and this may not suit your need. In that case, you can either set spring.jpa.database to default to let Hibernate figure things out or set the dialect using the spring.jpa.database-platform property.

The most common options to set are:

spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy spring.jpa.show-sql=true

In addition all properties in spring.jpa.properties.\* are passed through as normal JPA properties (with the prefix stripped) when the local EntityManagerFactory is created.

## Configure Hibernate Naming Strategy

Spring Boot provides a consistent naming strategy regardless of the Hibernate generation that you are using. If you are using Hibernate 4, you can customize it using spring.jpa.hibernate.naming.strategy; Hibernate 5 defines a Physical and Implicit naming strategies.

Spring Boot configures SpringPhysicalNamingStrategy by default. This implementation provides the same table structure as Hibernate 4: all dots are replaced by underscores and camel cases are replaced by underscores as well. By default, all table names are generated in lower case but it is possible to override that flag if your schema requires it.

Concretely, a TelephoneNumber entity will be mapped to the telephone\_number table. If you’d rather use Hibernate 5’s default instead, set the following property:

spring.jpa.hibernate.naming.physical- strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

See [HibernateJpaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.java) and [JpaBaseConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/JpaBaseConfiguration.java) for more details.

## Use a custom EntityManagerFactory

To take full control of the configuration of the EntityManagerFactory, you need to add a @Bean named ‘entityManagerFactory’. Spring Boot auto-configuration switches off its entity manager based on the presence of a bean of that type.

## Use Two EntityManagers

Even if the default EntityManagerFactory works fine, you will need to define a new one because otherwise the presence of the second bean of that type will switch off the default. To make it easy to do that you can use the convenient EntityManagerBuilder provided by Spring Boot, or if you prefer you can just use the LocalContainerEntityManagerFactoryBean directly from Spring ORM.

Example:

*// add two data sources configured as above*

@Bean

**public** LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(

EntityManagerFactoryBuilder builder) {

**return** builder

.dataSource(customerDataSource())

.packages(Customer.**class**)

.persistenceUnit(***"customers"***)

.build();

}

@Bean

**public** LocalContainerEntityManagerFactoryBean orderEntityManagerFactory( EntityManagerFactoryBuilder builder) {

**return** builder

.dataSource(orderDataSource())

.packages(Order.**class**)

.persistenceUnit(***"orders"***)

.build();

}

The configuration above almost works on its own. To complete the picture you need to configure TransactionManagers for the two EntityManagers as well. One of them could be picked up by the default JpaTransactionManager in Spring Boot if you mark it as @Primary. The other would have to be explicitly injected into a new instance. Or you might be able to use a JTA transaction manager spanning both.

If you are using Spring Data, you need to configure @EnableJpaRepositories accordingly:

@Configuration

@EnableJpaRepositories(basePackageClasses = Customer.class, entityManagerFactoryRef = "customerEntityManagerFactory")

**public class** CustomerConfiguration {

...

}

@Configuration

@EnableJpaRepositories(basePackageClasses = Order.class, entityManagerFactoryRef = "orderEntityManagerFactory")

**public class** OrderConfiguration {

...

}

## Use a traditional persistence.xml

Spring doesn’t require the use of XML to configure the JPA provider, and Spring Boot assumes you want to take advantage of that feature. If you prefer to use persistence.xml then you need to define your own @Bean of type LocalEntityManagerFactoryBean (with id ‘entityManagerFactory’, and set the persistence unit name there.

See [JpaBaseConfiguration](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/JpaBaseConfiguration.java) for the default settings.

## Use Spring Data JPA and Mongo repositories

Spring Data JPA and Spring Data Mongo can both create Repository implementations for you automatically. If they are both present on the classpath, you might have to do some extra configuration to tell Spring Boot which one (or both) you want to create repositories for you. The most explicit way to do that is to use the standard Spring Data @Enable\*Repositories and tell it the location of your Repository interfaces (where ‘\*’ is ‘Jpa’ or ‘Mongo’ or both).

There are also flags spring.data.\*.repositories.enabled that you can use to switch the auto- configured repositories on and off in external configuration. This is useful for instance in case you want to switch off the Mongo repositories and still use the auto-configured MongoTemplate.

The same obstacle and the same features exist for other auto-configured Spring Data repository types (Elasticsearch, Solr). Just change the names of the annotations and flags respectively.

## Expose Spring Data repositories as REST endpoint

Spring Data REST can expose the Repository implementations as REST endpoints for you as long as Spring MVC has been enabled for the application.

Spring Boot exposes a set of useful properties from the spring.data.rest namespace that customize the [RepositoryRestConfiguration](http://docs.spring.io/spring-data/rest/docs/current/api/org/springframework/data/rest/core/config/RepositoryRestConfiguration.html). If you need to provide additional customization, you should use a [RepositoryRestConfigurer](http://docs.spring.io/spring-data/rest/docs/current/api/org/springframework/data/rest/webmvc/config/RepositoryRestConfigurer.html) bean.

**Note**

If you don’t specify any order on your custom RepositoryRestConfigurer it will run after the one Spring Boot uses internally. If you need to specify an order, make sure it is higher than 0.

## Configure a component that is used by JPA

If you want to configure a component that will be used by JPA then you need to ensure that the component is initialized before JPA. Where the component is auto-configured Spring Boot will take care of this for you. For example, when Flyway is auto-configured, Hibernate is configured to depend upon Flyway so that the latter has a chance to initialize the database before Hibernate tries to use it.

If you are configuring a component yourself, you can use an EntityManagerFactoryDependsOnPostProcessor subclass as a convenient way of setting up the necessary dependencies. For example, if you are using Hibernate Search with Elasticsearch as its index manager then any EntityManagerFactory beans must be configured to depend on the elasticsearchClient bean:

**/\*\***

* **{@link EntityManagerFactoryDependsOnPostProcessor} that ensures that**
* **{@link EntityManagerFactory} beans depend on the {@code elasticsearchClient} bean.**

**\*/**

@Configuration

**static class** ElasticsearchJpaDependencyConfiguration

**extends** EntityManagerFactoryDependsOnPostProcessor {

ElasticsearchJpaDependencyConfiguration() {

**super**(***"elasticsearchClient"***);

}

}

# Database initialization

An SQL database can be initialized in different ways depending on what your stack is. Or of course you can do it manually as long as the database is a separate process.

## Initialize a database using JPA

JPA has features for DDL generation, and these can be set up to run on startup against the database. This is controlled through two external properties:

* spring.jpa.generate-ddl (boolean) switches the feature on and off and is vendor independent.
* spring.jpa.hibernate.ddl-auto (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. See below for more detail.

## Initialize a database using Hibernate

You can set spring.jpa.hibernate.ddl-auto explicitly and the standard Hibernate property values are none, validate, update, create, create-drop. Spring Boot chooses a default value for you based on whether it thinks your database is embedded (default create-drop) or not (default none). An embedded database is detected by looking at the Connection type: hsqldb, h2 and derby are embedded, the rest are not. Be careful when switching from in-memory to a ‘real’ database that you don’t make assumptions about the existence of the tables and data in the new platform. You either have to set ddl-auto explicitly, or use one of the other mechanisms to initialize the database.

**Note**

You can output the schema creation by enabling the org.hibernate.SQL logger. This is done for you automatically if you enable the [debug mode](#_bookmark121).

In addition, a file named import.sql in the root of the classpath will be executed on startup if Hibernate creates the schema from scratch (that is if the ddl-auto property is set to create or create-drop). This can be useful for demos and for testing if you are careful, but probably not something you want to be on the classpath in production. It is a Hibernate feature (nothing to do with Spring).

## Initialize a database

Spring Boot can automatically create the schema (DDL scripts) of your DataSource and initialize it (DML scripts): it loads SQL from the standard root classpath locations schema.sql and data.sql, respectively. In addition Spring Boot will process the schema-

${platform}.sql and data-${platform}.sql files (if present), where platform is the value of

spring.datasource.platform. This allows you to switch to database specific scripts if necessary,

e.g. you might choose to set it to the vendor name of the database (hsqldb, h2, oracle, mysql, postgresql etc.).

Spring Boot enables the fail-fast feature of the Spring JDBC initializer by default, so if the scripts cause exceptions the application will fail to start. You can tune that using spring.datasource.continue- on-error.

**Note**

In a JPA-based app, you can choose to let Hibernate create the schema or use schema.sql but not both. Make sure to disable spring.jpa.hibernate.ddl-auto if you chose the later.

You can also disable initialization by setting spring.datasource.initialize to false.

## Initialize a Spring Batch database

If you are using Spring Batch then it comes pre-packaged with SQL initialization scripts for most popular database platforms. Spring Boot will detect your database type, and execute those scripts by default, and in this case will switch the fail fast setting to false (errors are logged but do not prevent the application from starting). This is because the scripts are known to be reliable and generally do not contain bugs, so errors are ignorable, and ignoring them makes the scripts idempotent. You can switch off the initialization explicitly using spring.batch.initializer.enabled=false.

## Use a higher-level database migration tool

Spring Boot supports two higher-level migration tools: [Flyway](http://flywaydb.org/) and [Liquibase](http://www.liquibase.org/).

### Execute Flyway database migrations on startup

To automatically run Flyway database migrations on startup, add the org.flywaydb:flyway-core

to your classpath.

The migrations are scripts in the form V<VERSION> <NAME>.sql (with <VERSION> an underscore- separated version, e.g. ‘1’ or ‘2\_1’). By default they live in a folder classpath:db/migration but you can modify that using flyway.locations. You can also add a special {vendor} placeholder to use vendor-specific scripts. Assume the following:

**flyway.locations**=db/migration/{vendor}

Rather than using db/migration, this configuration will set the folder to use according to the type of the database (i.e. db/migration/mysql for MySQL). The list of supported database are available in [DatabaseDriver](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jdbc/DatabaseDriver.java).

See also the Flyway class from flyway-core for details of available settings like schemas etc. In addition Spring Boot provides a small set of properties in [FlywayProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayProperties.java) that can be used to disable the migrations, or switch off the location checking. Spring Boot will call Flyway.migrate() to perform the database migration. If you would like more control, provide a @Bean that implements [FlywayMigrationStrategy](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayMigrationStrategy.java).

**Tip**

If you want to make use of [Flyway callbacks](http://flywaydb.org/documentation/callbacks.html), those scripts should also live in the classpath:db/ migration folder.

By default Flyway will autowire the (@Primary) DataSource in your context and use that for migrations. If you like to use a different DataSource you can create one and mark its @Bean as @FlywayDataSource - if you do that remember to create another one and mark it as @Primary if you want two data sources. Or you can use Flyway’s native DataSource by setting flyway. [url,user,password] in external properties.

There is a [Flyway sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-flyway) so you can see how to set things up.

You can also use Flyway to provide data for specific scenarios. For example, you can place test- specific migrations in src/test/resources and they will only be run when your application starts for testing. If you want to be more sophisticated you can use profile-specific configuration to customize flyway.locations so that certain migrations will only run when a particular profile is active. For example, in application-dev.properties you could set flyway.locations to classpath:/ db/migration, classpath:/dev/db/migration and migrations in dev/db/migration will only run when the dev profile is active.

### Execute Liquibase database migrations on startup

To automatically run Liquibase database migrations on startup, add the

org.liquibase:liquibase-core to your classpath.

The master change log is by default read from db/changelog/db.changelog-master.yaml but can be set using liquibase.change-log. In addition to YAML, Liquibase also supports JSON, XML, and SQL change log formats.

By default Liquibase will autowire the (@Primary) DataSource in your context and use that for migrations. If you like to use a different DataSource you can create one and mark its @Bean as @LiquibaseDataSource - if you do that remember to create another one and mark it as @Primary if you want two data sources. Or you can use Liquibase’s native DataSource by setting liquibase. [url,user,password] in external properties.

See [LiquibaseProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/liquibase/LiquibaseProperties.java) for details of available settings like contexts, default schema etc. There is a [Liquibase sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-liquibase) so you can see how to set things up.

# Messaging

## Disable transacted JMS session

If your JMS broker does not support transacted session, you will have to disable the support of transactions altogether. If you create your own JmsListenerContainerFactory there is nothing to do since it won’t be transacted by default. If you want to use the DefaultJmsListenerContainerFactoryConfigurer to reuse Spring Boot’s default, you can disable transacted session as follows:

@Bean

**public** DefaultJmsListenerContainerFactory jmsListenerContainerFactory( ConnectionFactory connectionFactory, DefaultJmsListenerContainerFactoryConfigurer configurer) {

DefaultJmsListenerContainerFactory listenerFactory =

**new** DefaultJmsListenerContainerFactory(); configurer.configure(listenerFactory, connectionFactory); listenerFactory.setTransactionManager(null); listenerFactory.setSessionTransacted(false);

**return** listenerFactory;

}

This overrides the default factory and this should be applied to any other factory that your application defines, if any.

# Batch applications

**Note**

By default, batch applications require a DataSource to store job details. If you want to deviate from that, you’ll need to implement BatchConfigurer, see [The Javadoc of](http://docs.spring.io/spring-batch/apidocs/org/springframework/batch/core/configuration/annotation/EnableBatchProcessing.html) [@EnableBatchProcessing](http://docs.spring.io/spring-batch/apidocs/org/springframework/batch/core/configuration/annotation/EnableBatchProcessing.html) for more details.

## Execute Spring Batch jobs on startup

Spring Batch auto-configuration is enabled by adding @EnableBatchProcessing (from Spring Batch) somewhere in your context.

By default it executes **all** Jobs in the application context on startup (see [JobLauncherCommandLineRunner](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/JobLauncherCommandLineRunner.java) for details). You can narrow down to a specific job or jobs by specifying spring.batch.job.names (comma-separated job name patterns).

If the application context includes a JobRegistry then the jobs in spring.batch.job.names are looked up in the registry instead of being autowired from the context. This is a common pattern with more complex systems where multiple jobs are defined in child contexts and registered centrally.

See [BatchAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/BatchAutoConfiguration.java) and [@EnableBatchProcessing](https://github.com/spring-projects/spring-batch/blob/master/spring-batch-core/src/main/java/org/springframework/batch/core/configuration/annotation/EnableBatchProcessing.java) for more details.

# Actuator

## Change the HTTP port or address of the actuator endpoints

In a standalone application the Actuator HTTP port defaults to the same as the main HTTP port. To make the application listen on a different port set the external property management.port. To listen on a completely different network address (e.g. if you have an internal network for management and an external one for user applications) you can also set management.address to a valid IP address that the server is able to bind to.

For more detail look at the [ManagementServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ManagementServerProperties.java) source code and [*Section 48.3,*](#_bookmark317)[*“Customizing the management server port”*](#_bookmark317) in the ‘Production-ready features’ section.

## Customize the ‘whitelabel’ error page

Spring Boot installs a ‘whitelabel’ error page that you will see in browser client if you encounter a server error (machine clients consuming JSON and other media types should see a sensible response with the right error code).

**Note**

Set server.error.whitelabel.enabled=false to switch the default error page off which will restore the default of the servlet container that you are using. Note that Spring Boot will still attempt to resolve the error view so you’d probably add you own error page rather than disabling it completely.

Overriding the error page with your own depends on the templating technology that you are using. For example, if you are using Thymeleaf you would add an error.html template and if you are using FreeMarker you would add an error.ftl template. In general what you need is a View that resolves with a name of error, and/or a @Controller that handles the /error path. Unless you replaced some of the default configuration you should find a BeanNameViewResolver in your ApplicationContext so a @Bean with id error would be a simple way of doing that. Look at [ErrorMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration.java) for more options.

See also the section on [Error Handling](#_bookmark139) for details of how to register handlers in the servlet container.

## Actuator and Jersey

Actuator HTTP endpoints are only available for Spring MVC-based applications. If you want to use Jersey and still use the actuator you will need to enable Spring MVC (by depending on spring- boot-starter-web, for example). By default, both Jersey and the Spring MVC dispatcher servlet are mapped to the same path (/). You will need to change the path for one of them (by configuring server.servlet-path for Spring MVC or spring.jersey.application-path for Jersey). For example, if you add server.servlet-path=/system into application.properties, the actuator HTTP endpoints will be available under /system.

# Security

## Switch off the Spring Boot security configuration

If you define a @Configuration with @EnableWebSecurity anywhere in your application it will switch off the default webapp security settings in Spring Boot (but leave the Actuator’s security enabled). To tweak the defaults try setting properties in security.\* (see [SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java) for details of available settings) and SECURITY section of [Common application properties](#_bookmark570).

## Change the AuthenticationManager and add user accounts

If you provide a @Bean of type AuthenticationManager the default one will not be created, so you have the full feature set of Spring Security available (e.g. [various authentication options](http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#jc-authentication)).

Spring Security also provides a convenient AuthenticationManagerBuilder which can be used to build an AuthenticationManager with common options. The recommended way to use this in a webapp is to inject it into a void method in a WebSecurityConfigurerAdapter, e.g.

@Configuration

**public class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {

@Autowired

**public void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception { auth.inMemoryAuthentication()

.withUser(***"barry"***).password(***"password"***).roles(***"USER"***); *// ... etc.*

}

*// ... other stuff for application security*

}

You will get the best results if you put this in a nested class, or a standalone class (i.e. not mixed in with a lot of other @Beans that might be allowed to influence the order of instantiation). The [secure web](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-web-secure) [sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-web-secure) is a useful template to follow.

If you experience instantiation issues (e.g. using JDBC or JPA for the user detail store) it might be worth extracting the AuthenticationManagerBuilder callback into a GlobalAuthenticationConfigurerAdapter (in the init() method so it happens before the authentication manager is needed elsewhere), e.g.

@Configuration

**public class** AuthenticationManagerConfiguration **extends**

GlobalAuthenticationConfigurerAdapter {

@Override

**public void** init(AuthenticationManagerBuilder auth) { auth.inMemoryAuthentication() *// ... etc.*

}

}

## Enable HTTPS when running behind a proxy server

Ensuring that all your main endpoints are only available over HTTPS is an important chore for any application. If you are using Tomcat as a servlet container, then Spring Boot will add Tomcat’s own

RemoteIpValve automatically if it detects some environment settings, and you should be able to rely on the HttpServletRequest to report whether it is secure or not (even downstream of a proxy server that handles the real SSL termination). The standard behavior is determined by the presence or absence of certain request headers (x-forwarded-for and x-forwarded-proto), whose names are conventional, so it should work with most front end proxies. You can switch on the valve by adding some entries to application.properties, e.g.

**server.tomcat.remote-ip-header**=x-forwarded-for **server.tomcat.protocol-header**=x-forwarded-proto

(The presence of either of those properties will switch on the valve. Or you can add the RemoteIpValve

yourself by adding a TomcatEmbeddedServletContainerFactory bean.)

Spring Security can also be configured to require a secure channel for all (or some requests). To switch that on in a Spring Boot application you just need to set security.require\_ssl to true in application.properties.

# Hot swapping

## Reload static content

There are several options for hot reloading. The recommended approach is to use [spring-boot-](#_bookmark67) [devtools](#_bookmark67) as it provides additional development-time features such as support for fast application restarts and LiveReload as well as sensible development-time configuration (e.g. template caching). Devtools works by monitoring the classpath for changes. This means that static resource changes must be "built" for the change to take affect. By default, this happens automatically in Eclipse when you save your changes. In IntelliJ IDEA, Make Project will trigger the necessary build. Due to the [default restart](#_bookmark71) [exclusions](#_bookmark71), changes to static resources will not trigger a restart of your application. They will, however, trigger a live reload.

Alternatively, running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also hot-swapping of Java class changes).

Finally, the [Maven and Gradle plugins](#_bookmark407) can be configured (see the addResources property) to support running from the command line with reloading of static files directly from source. You can use that with an external css/js compiler process if you are writing that code with higher level tools.

## Reload templates without restarting the container

Most of the templating technologies supported by Spring Boot include a configuration option to disable caching (see below for details). If you’re using the spring-boot-devtools module these properties will be [automatically configured](#_bookmark68) for you at development time.

### Thymeleaf templates

If you are using Thymeleaf, then set spring.thymeleaf.cache to false. See

[ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java) for other Thymeleaf customization options.

### FreeMarker templates

If you are using FreeMarker, then set spring.freemarker.cache to false. See

[FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java) for other FreeMarker customization options.

### Groovy templates

If you are using Groovy templates, then set spring.groovy.template.cache to false. See

[GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java) for other Groovy customization options.

## Fast application restarts

The spring-boot-devtools module includes support for automatic application restarts. Whilst not as fast as technologies such as [JRebel](http://zeroturnaround.com/software/jrebel/) or [Spring Loaded](https://github.com/spring-projects/spring-loaded) it’s usually significantly faster than a “cold start”. You should probably give it a try before investigating some of the more complex reload options discussed below.

For more details see the [Chapter 20, *Developer tools*](#_bookmark67) section.

## Reload Java classes without restarting the container

Modern IDEs (Eclipse, IDEA, etc.) all support hot swapping of bytecode, so if you make a change that doesn’t affect class or method signatures it should reload cleanly with no side effects.

[Spring Loaded](https://github.com/spring-projects/spring-loaded) goes a little further in that it can reload class definitions with changes in the method signatures. With some customization it can force an ApplicationContext to refresh itself (but there is no general mechanism to ensure that would be safe for a running application anyway, so it would only ever be a development time trick probably).

### Configuring Spring Loaded for use with Maven

To use Spring Loaded with the Maven command line, just add it as a dependency in the Spring Boot plugin declaration, e.g.

**<plugin>**

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

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

**<dependencies>**

**<dependency>**

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

**<artifactId>**springloaded**</artifactId>**

**<version>**1.2.6.RELEASE**</version>**

**</dependency>**

**</dependencies>**

**</plugin>**

This normally works pretty well with Eclipse and IntelliJ IDEA as long as they have their build configuration aligned with the Maven defaults (Eclipse m2e does this out of the box).

### Configuring Spring Loaded for use with Gradle and IntelliJ IDEA

You need to jump through a few hoops if you want to use Spring Loaded in combination with Gradle and IntelliJ IDEA. By default, IntelliJ IDEA will compile classes into a different location than Gradle, causing Spring Loaded monitoring to fail.

To configure IntelliJ IDEA correctly you can use the idea Gradle plugin:

buildscript {

repositories { jcenter() } dependencies {

classpath ***"org.springframework.boot:spring-boot-gradle-plugin:1.5.8.RELEASE"***

classpath ***'org.springframework:springloaded:1.2.6.RELEASE'***

}

}

apply plugin: ***'idea'***

idea {

module {

inheritOutputDirs = false

outputDir = file(***"$buildDir/classes/main/"***)

}

}

*// ...*

**Note**

IntelliJ IDEA must be configured to use the same Java version as the command line Gradle task and springloaded **must** be included as a buildscript dependency.

You can also additionally enable ‘Make Project Automatically’ inside IntelliJ IDEA to automatically compile your code whenever a file is saved.

# Build

## Generate build information

Both the Maven and Gradle plugin allow to generate build information containing the coordinates, name and version of the project. The plugin can also be configured to add additional properties through configuration. When such file is present, Spring Boot auto-configures a BuildProperties bean.

To generate build information with Maven, add an execution for the build-info goal:

**<build>**

**<plugins>**

**<plugin>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**<executions>**

**<execution>**

**<goals>**

**<goal>**build-info**</goal>**

**</goals>**

**</execution>**

**</executions>**

**</plugin>**

**</plugins>**

**</build>**

**Tip**

Check the [Spring Boot Maven Plugin documentation](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/) for more details. And to do the same with Gradle:

springBoot {

buildInfo()

}

Additional properties can be added using the DSL:

springBoot {

buildInfo {

additionalProperties = [

***'foo'***: ***'bar'***

]

}

}

## Generate git information

Both Maven and Gradle allow to generate a git.properties file containing information about the state of your git source code repository when the project was built.

For Maven users the spring-boot-starter-parent POM includes a pre-configured plugin to generate a git.properties file. Simply add the following declaration to your POM:

**<build>**

**<plugins>**

**<plugin>**

**<groupId>**pl.project13.maven**</groupId>**

**<artifactId>**git-commit-id-plugin**</artifactId>**

**</plugin>**

**</plugins>**

**</build>**

Gradle users can achieve the same result using the [gradle-git-properties](https://plugins.gradle.org/plugin/com.gorylenko.gradle-git-properties) plugin

plugins {

id ***"com.gorylenko.gradle-git-properties"*** version ***"1.4.17"***

}

**Tip**

The commit time in git.properties is expected to match the format yyyy-MM- dd’T’HH:mm:ssZ. This is the default format for both plugins listed above. Using this format allows the time to be parsed into a Date and its format when serialized to JSON to be controlled by Jackson’s date serialization configuration settings.

## Customize dependency versions

If you use a Maven build that inherits directly or indirectly from spring-boot-dependencies (for instance spring-boot-starter-parent) but you want to override a specific third-party dependency you can add appropriate <properties> elements. Browse the [spring-boot-dependencies](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-dependencies/pom.xml) POM for a complete list of properties. For example, to pick a different slf4j version you would add the following:

**<properties>**

**<slf4j.version>**1.7.5**<slf4j.version>**

**</properties>**

**Note**

This only works if your Maven project inherits (directly or indirectly) from spring- boot-dependencies. If you have added spring-boot-dependencies in your own dependencyManagement section with <scope>import</scope> you have to redefine the artifact yourself instead of overriding the property.

**Warning**

Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues.

To override dependency versions in Gradle, you can specify a version as shown below:

ext[***'slf4j.version'***] = ***'1.7.5'***

For additional information, please refer to the [Gradle Dependency Management Plugin documentation](https://github.com/spring-gradle-plugins/dependency-management-plugin).

## Create an executable JAR with Maven

The spring-boot-maven-plugin can be used to create an executable ‘fat’ JAR. If you are using the spring-boot-starter-parent POM you can simply declare the plugin and your jars will be repackaged:

**<build>**

**<plugins>**

**<plugin>**

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

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

**</plugin>**

**</plugins>**

**</build>**

If you are not using the parent POM you can still use the plugin, however, you must additionally add an <executions> section:

**<build>**

**<plugins>**

**<plugin>**

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

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

**<version>**1.5.8.RELEASE**</version>**

**<executions>**

**<execution>**

**<goals>**

**<goal>**repackage**</goal>**

**</goals>**

**</execution>**

**</executions>**

**</plugin>**

**</plugins>**

**</build>**

See the [plugin documentation](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/usage.html) for full usage details.

## Use a Spring Boot application as a dependency

Like a war file, a Spring Boot application is not intended to be used as a dependency. If your application contains classes that you want to share with other projects, the recommended approach is to move that code into a separate module. The separate module can then be depended upon by your application and other projects.

If you cannot rearrange your code as recommended above, Spring Boot’s Maven and Gradle plugins must be configured to produce a separate artifact that is suitable for use as a dependency. The executable archive cannot be used as a dependency as the [executable jar format](#_bookmark595) packages application classes in BOOT-INF/classes. This means that they cannot be found when the executable jar is used as a dependency.

To produce the two artifacts, one that can be used as a dependency and one that is executable, a classifier must be specified. This classifier is applied to the name of the executable archive, leaving the default archive for use as dependency.

To configure a classifier of exec in Maven, the following configuration can be used:

**<build>**

**<plugins>**

**<plugin>**

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

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

**<configuration>**

**<classifier>**exec**</classifier>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

And when using Gradle, the following configuration can be used:

bootRepackage { classifier = ***'exec'***

}

## Extract specific libraries when an executable jar runs

Most nested libraries in an executable jar do not need to be unpacked in order to run, however, certain libraries can have problems. For example, JRuby includes its own nested jar support which assumes that the jruby-complete.jar is always directly available as a file in its own right.

To deal with any problematic libraries, you can flag that specific nested jars should be automatically unpacked to the ‘temp folder’ when the executable jar first runs.

For example, to indicate that JRuby should be flagged for unpack using the Maven Plugin you would add the following configuration:

**<build>**

**<plugins>**

**<plugin>**

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

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

**<configuration>**

**<requiresUnpack>**

**<dependency>**

**<groupId>**org.jruby**</groupId>**

**<artifactId>**jruby-complete**</artifactId>**

**</dependency>**

**</requiresUnpack>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

And to do that same with Gradle:

springBoot {

requiresUnpack = [***'org.jruby:jruby-complete'***]

}

## Create a non-executable JAR with exclusions

Often if you have an executable and a non-executable jar as build products, the executable version will have additional configuration files that are not needed in a library jar. E.g. the application.yml configuration file might excluded from the non-executable JAR.

Here’s how to do that in Maven:

**<build>**

**<plugins>**

**<plugin>**

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

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

**<configuration>**

**<classifier>**exec**</classifier>**

**</configuration>**

**</plugin>**

**<plugin>**

**<artifactId>**maven-jar-plugin**</artifactId>**

**<executions>**

**<execution>**

**<id>**exec**</id>**

**<phase>**package**</phase>**

**<goals>**

**<goal>**jar**</goal>**

**</goals>**

**<configuration>**

**<classifier>**exec**</classifier>**

**</configuration>**

**</execution>**

**<execution>**

**<phase>**package**</phase>**

**<goals>**

**<goal>**jar**</goal>**

**</goals>**

**<configuration>**

*<!-- Need this to ensure application.yml is excluded -->*

**<forceCreation>**true**</forceCreation>**

**<excludes>**

**<exclude>**application.yml**</exclude>**

**</excludes>**

**</configuration>**

**</execution>**

**</executions>**

**</plugin>**

**</plugins>**

**</build>**

In Gradle you can create a new JAR archive with standard task DSL features, and then have the

bootRepackage task depend on that one using its withJarTask property:

jar {

baseName = ***'spring-boot-sample-profile'***

version = ***'0.0.0'***

excludes = [***'\*\*/application.yml'***]

}

task(***'execJar'***, type:Jar, dependsOn: ***'jar'***) { baseName = ***'spring-boot-sample-profile'*** version = ***'0.0.0'***

classifier = ***'exec'***

from sourceSets.main.output

}

bootRepackage {

withJarTask = tasks[***'execJar'***]

}

## Remote debug a Spring Boot application started with Maven

To attach a remote debugger to a Spring Boot application started with Maven you can use the

jvmArguments property of the [maven plugin](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/). Check [this example](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/maven-plugin/examples/run-debug.html) for more details.

## Remote debug a Spring Boot application started with Gradle

To attach a remote debugger to a Spring Boot application started with Gradle you can use the jvmArgs

property of bootRun task or --debug-jvm command line option.

build.gradle:

bootRun {

jvmArgs ***"-agentlib:jdwp=transport=dt\_socket,server=y,suspend=y,address=5005"***

}

Command line:

$ gradle bootRun --debug-jvm

Check [Gradle Application Plugin](http://www.gradle.org/docs/current/userguide/application_plugin.html) for more details.

## Build an executable archive from Ant without using spring-boot-antlib

To build with Ant you need to grab dependencies, compile and then create a jar or war archive. To make it executable you can either use the spring-boot-antlib module, or you can follow these instructions:

1. If you are building a jar, package the application’s classes and resources in a nested BOOT-INF/ classes directory. If you are building a war, package the application’s classes in a nested WEB- INF/classes directory as usual.
2. Add the runtime dependencies in a nested BOOT-INF/lib directory for a jar or WEB-INF/lib for a war. Remember **not** to compress the entries in the archive.
3. Add the provided (embedded container) dependencies in a nested BOOT-INF/lib directory for jar or WEB-INF/lib-provided for a war. Remember **not** to compress the entries in the archive.
4. Add the spring-boot-loader classes at the root of the archive (so the Main-Class is available).
5. Use the appropriate launcher, e.g. JarLauncher for a jar file, as a Main-Class attribute in the manifest and specify the other properties it needs as manifest entries, principally a Start-Class.

Example:

**<target name**=**"build" depends**=**"compile">**

**<jar destfile**=**"target/${ant.project.name}-${spring-boot.version}.jar" compress**=**"false">**

**<mappedresources>**

**<fileset dir**=**"target/classes" />**

**<globmapper from**=**"\*" to**=**"BOOT-INF/classes/\*"/>**

**</mappedresources>**

**<mappedresources>**

**<fileset dir**=**"src/main/resources" erroronmissingdir**=**"false"/>**

**<globmapper from**=**"\*" to**=**"BOOT-INF/classes/\*"/>**

**</mappedresources>**

**<mappedresources>**

**<fileset dir**=**"${lib.dir}/runtime" />**

**<globmapper from**=**"\*" to**=**"BOOT-INF/lib/\*"/>**

**</mappedresources>**

**<zipfileset src**=**"${lib.dir}/loader/spring-boot-loader-jar-${spring-boot.version}.jar" />**

**<manifest>**

**<attribute name**=**"Main-Class" value**=**"org.springframework.boot.loader.JarLauncher" />**

**<attribute name**=**"Start-Class" value**=**"${start-class}" />**

**</manifest>**

**</jar>**

**</target>**

The [Ant Sample](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-ant) has a build.xml with a manual task that should work if you run it with

$ ant -lib <folder containing ivy-2.2.jar> clean manual

after which you can run the application with

$ java -jar target/\*.jar

## How to use Java 6

If you want to use Spring Boot with Java 6 there are a small number of configuration changes that you will have to make. The exact changes depend on your application’s functionality.

### Embedded servlet container compatibility

If you are using one of Boot’s embedded Servlet containers you will have to use a Java 6-compatible container. Both Tomcat 7 and Jetty 8 are Java 6 compatible. See [Section 73.16, “Use Tomcat 7.x or](#_bookmark476) [8.0”](#_bookmark476) and [Section 73.18, “Use Jetty 8”](#_bookmark482) for details.

### Jackson

Jackson 2.7 and later requires Java 7. If you want to use Jackson with Java 6 you will have to downgrade to Jackson 2.6.

Spring Boot uses the Jackson BOM that was introduced as of Jackson 2.7 so you can’t just override the jackson.version property. In order to use Jackson 2.6, you will have to define the individual modules in the dependencyManagement section of your build, check [this example](https://github.com/spring-projects/spring-boot/blob/0ffc7dc13f6de82c199a6d503354a88c7aaec2d9/spring-boot-dependencies/pom.xml#L523-L597) for more details.

### JTA API compatibility

While the Java Transaction API itself doesn’t require Java 7 the official API jar contains classes that have been built to require Java 7. If you are using JTA then you will need to replace the official JTA 1.2 API jar with one that has been built to work on Java 6. To do so, exclude any transitive dependencies on javax.transaction:javax.transaction-api and replace them with a dependency on org.jboss.spec.javax.transaction:jboss-transaction-api\_1.2\_spec:1.0.0.Final

# Traditional deployment

## Create a deployable war file

The first step in producing a deployable war file is to provide a SpringBootServletInitializer subclass and override its configure method. This makes use of Spring Framework’s Servlet 3.0 support and allows you to configure your application when it’s launched by the servlet container. Typically, you update your application’s main class to extend SpringBootServletInitializer:

@SpringBootApplication

**public class** Application **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(Application.**class**);

}

**public static void** main(String[] args) **throws** Exception { SpringApplication.run(Application.**class**, args);

}

}

The next step is to update your build configuration so that your project produces a war file rather than a jar file. If you’re using Maven and using spring-boot-starter-parent (which configures Maven’s war plugin for you) all you need to do is to modify pom.xml to change the packaging to war:

**<packaging>**war**</packaging>**

If you’re using Gradle, you need to modify build.gradle to apply the war plugin to the project:

apply plugin: ***'war'***

The final step in the process is to ensure that the embedded servlet container doesn’t interfere with the servlet container to which the war file will be deployed. To do so, you need to mark the embedded servlet container dependency as provided.

If you’re using Maven:

**<dependencies>**

*<!-- … -->*

**<dependency>**

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

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

**<scope>**provided**</scope>**

**</dependency>**

*<!-- … -->*

**</dependencies>**

And if you’re using Gradle:

dependencies {

*// …*

providedRuntime ***'org.springframework.boot:spring-boot-starter-tomcat'***

*// …*

}

**Note**

If you are using a version of Gradle that supports compile only dependencies (2.12 or later), you should continue to use providedRuntime. Among other limitations, compileOnly dependencies are not on the test classpath so any web-based integration tests will fail.

If you’re using the [Spring Boot build tools](#_bookmark407), marking the embedded servlet container dependency as provided will produce an executable war file with the provided dependencies packaged in a lib- provided directory. This means that, in addition to being deployable to a servlet container, you can also run your application using java -jar on the command line.

**Tip**

Take a look at Spring Boot’s sample applications for a [Maven-based example](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-samples/spring-boot-sample-traditional/pom.xml) of the above- described configuration.

## Create a deployable war file for older servlet containers

Older Servlet containers don’t have support for the ServletContextInitializer bootstrap process used in Servlet 3.0. You can still use Spring and Spring Boot in these containers but you are going to need to add a web.xml to your application and configure it to load an ApplicationContext via a DispatcherServlet.

## Convert an existing application to Spring Boot

For a non-web application it should be easy (throw away the code that creates your ApplicationContext and replace it with calls to SpringApplication or SpringApplicationBuilder). Spring MVC web applications are generally amenable to first creating a deployable war application, and then migrating it later to an executable war and/or jar. Useful reading is in the [Getting Started Guide on Converting a jar to a war](http://spring.io/guides/gs/convert-jar-to-war/).

Create a deployable war by extending SpringBootServletInitializer (e.g. in a class called

Application), and add the Spring Boot @SpringBootApplication annotation. Example:

@SpringBootApplication

**public class** Application **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

*// Customize the application or call application.sources(...) to add sources*

*// Since our example is itself a @Configuration class (via @SpringBootApplication)*

*// we actually don't need to override this method.*

**return** application;

}

}

Remember that whatever you put in the sources is just a Spring ApplicationContext and normally anything that already works should work here. There might be some beans you can remove later and let Spring Boot provide its own defaults for them, but it should be possible to get something working first.

Static resources can be moved to /public (or /static or /resources or /META-INF/resources) in the classpath root. Same for messages.properties (Spring Boot detects this automatically in the root of the classpath).

Vanilla usage of Spring DispatcherServlet and Spring Security should require no further changes. If you have other features in your application, using other servlets or filters for instance, then you may need to add some configuration to your Application context, replacing those elements from the web.xml as follows:

* A @Bean of type Servlet or ServletRegistrationBean installs that bean in the container as if it was a <servlet/> and <servlet-mapping/> in web.xml.
* A @Bean of type Filter or FilterRegistrationBean behaves similarly (like a <filter/> and

<filter-mapping/>.

* An ApplicationContext in an XML file can be added through an @ImportResource in your Application. Or simple cases where annotation configuration is heavily used already can be recreated in a few lines as @Bean definitions.

Once the war is working we make it executable by adding a main method to our Application, e.g.

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

}

**Note**

If you intend to start your application as a war or as an executable application, you need to share the customizations of the builder in a method that is both available to the SpringBootServletInitializer callback and the main method, something like:

@SpringBootApplication

**public class** Application **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {

**return** configureApplication(builder);

}

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

configureApplication(**new** SpringApplicationBuilder()).run(args);

}

**private static** SpringApplicationBuilder configureApplication(SpringApplicationBuilder builder) {

**return** builder.sources(Application.**class**).bannerMode(Banner.Mode.OFF);

}

}

Applications can fall into more than one category:

* Servlet 3.0+ applications with no web.xml.
* Applications with a web.xml.
* Applications with a context hierarchy.
* Applications without a context hierarchy.

All of these should be amenable to translation, but each might require slightly different tricks.

Servlet 3.0+ applications might translate pretty easily if they already use the Spring Servlet 3.0+ initializer support classes. Normally all the code from an existing WebApplicationInitializer

can be moved into a SpringBootServletInitializer. If your existing application has more than one ApplicationContext (e.g. if it uses AbstractDispatcherServletInitializer) then you might be able to squash all your context sources into a single SpringApplication. The main complication you might encounter is if that doesn’t work and you need to maintain the context hierarchy. See the [entry on building a hierarchy](#_bookmark443) for examples. An existing parent context that contains web-specific features will usually need to be broken up so that all the ServletContextAware components are in the child context.

Applications that are not already Spring applications might be convertible to a Spring Boot application, and the guidance above might help, but your mileage may vary.

## Deploying a WAR to WebLogic

To deploy a Spring Boot application to WebLogic you must ensure that your servlet initializer **directly** implements WebApplicationInitializer (even if you extend from a base class that already implements it).

A typical initializer for WebLogic would be something like this:

**import** org.springframework.boot.autoconfigure.SpringBootApplication; **import** org.springframework.boot.context.web.SpringBootServletInitializer; **import** org.springframework.web.WebApplicationInitializer;

@SpringBootApplication

**public class** MyApplication **extends** SpringBootServletInitializer **implements** WebApplicationInitializer {

}

If you use logback, you will also need to tell WebLogic to prefer the packaged version rather than the version that pre-installed with the server. You can do this by adding a WEB-INF/weblogic.xml file with the following contents:

<?xml version="1.0" encoding="UTF-8"?>

**<wls:weblogic-web-app xmlns:wls**=[**"http://xmlns.oracle.com/weblogic/weblogic-web-app"**](http://xmlns.oracle.com/weblogic/weblogic-web-app) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://java.sun.com/xml/ns/javaee**](http://java.sun.com/xml/ns/javaee)

[**http://java.sun.com/xml/ns/javaee/ejb-jar\_3\_0.xsd**](http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd)[**http://xmlns.oracle.com/weblogic/weblogic-web-app**](http://xmlns.oracle.com/weblogic/weblogic-web-app)[**http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd"**](http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd)**>**

**<wls:container-descriptor>**

**<wls:prefer-application-packages>**

**<wls:package-name>**org.slf4j**</wls:package-name>**

**</wls:prefer-application-packages>**

**</wls:container-descriptor>**

**</wls:weblogic-web-app>**

## Deploying a WAR in an Old (Servlet 2.5) Container

Spring Boot uses Servlet 3.0 APIs to initialize the ServletContext (register Servlets etc.) so you can’t use the same application out of the box in a Servlet 2.5 container. It **is** however possible to run a Spring Boot application on an older container with some special tools. If you include org.springframework.boot:spring-boot-legacy as a dependency ([maintained separately](https://github.com/scratches/spring-boot-legacy) to the core of Spring Boot and currently available at 1.0.2.RELEASE), all you should need to do is create a web.xml and declare a context listener to create the application context and your filters and servlets. The context listener is a special purpose one for Spring Boot, but the rest of it is normal for a Spring application in Servlet 2.5. Example:

<?xml version="1.0" encoding="UTF-8"?>

**<web-app version**=**"2.5" xmlns**=[**"http://java.sun.com/xml/ns/javaee"**](http://java.sun.com/xml/ns/javaee) **xmlns:xsi**=[**"http://www.w3.org/2001/XMLSchema-instance"**](http://www.w3.org/2001/XMLSchema-instance) **xsi:schemaLocation**=[**"http://java.sun.com/xml/ns/javaee**](http://java.sun.com/xml/ns/javaee)[**http://java.sun.com/xml/ns/javaee/web-**](http://java.sun.com/xml/ns/javaee/web-)

**app\_2\_5.xsd">**

**<context-param>**

**<param-name>**contextConfigLocation**</param-name>**

**<param-value>**demo.Application**</param-value>**

**</context-param>**

**<listener>**

**<listener-class>**org.springframework.boot.legacy.context.web.SpringBootContextLoaderListener**</ listener-class>**

**</listener>**

**<filter>**

**<filter-name>**metricsFilter**</filter-name>**

**<filter-class>**org.springframework.web.filter.DelegatingFilterProxy**</filter-class>**

**</filter>**

**<filter-mapping>**

**<filter-name>**metricsFilter**</filter-name>**

**<url-pattern>**/\***</url-pattern>**

**</filter-mapping>**

**<servlet>**

**<servlet-name>**appServlet**</servlet-name>**

**<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**

**<init-param>**

**<param-name>**contextAttribute**</param-name>**

**<param-value>**org.springframework.web.context.WebApplicationContext.ROOT**</param-value>**

**</init-param>**

**<load-on-startup>**1**</load-on-startup>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>**appServlet**</servlet-name>**

**<url-pattern>**/**</url-pattern>**

**</servlet-mapping>**

**</web-app>**

In this example we are using a single application context (the one created by the context listener) and attaching it to the DispatcherServlet using an init parameter. This is normal in a Spring Boot application (you normally only have one application context).

**Part X. Appendices**

**Appendix A. Common application properties**

Various properties can be specified inside your application.properties/application.yml file or as command line switches. This section provides a list of common Spring Boot properties and references to the underlying classes that consume them.

**Note**

Property contributions can come from additional jar files on your classpath so you should not consider this an exhaustive list. It is also perfectly legit to define your own properties.

**Warning**

This sample file is meant as a guide only. Do **not** copy/paste the entire content into your application; rather pick only the properties that you need.

*# =================================================================== # COMMON SPRING BOOT PROPERTIES*

*#*

*# This sample file is provided as a guideline. Do NOT copy it in its # entirety to your own application. ^^^*

*# ===================================================================*

*# ---------------------------------------- # CORE PROPERTIES*

*# ----------------------------------------*

*# BANNER*

**banner.charset**=UTF-8 *# Banner file encoding.*

**banner.location**=classpath:banner.txt *# Banner file location.* **banner.image.location**=classpath:banner.gif *# Banner image file location (jpg/png can also be used).* **banner.image.width**= *# Width of the banner image in chars (default 76)*

**banner.image.height**= *# Height of the banner image in chars (default based on image height)*

**banner.image.margin**= *# Left hand image margin in chars (default 2)*

**banner.image.invert**= *# If images should be inverted for dark terminal themes (default false)*

*# LOGGING*

**logging.config**= *# Location of the logging configuration file. For instance `classpath:logback.xml` for Logback*

**logging.exception-conversion-word**=%wEx *# Conversion word used when logging exceptions.*

**logging.file**= *# Log file name. For instance `myapp.log`*

**logging.level.\***= *# Log levels severity mapping. For instance `logging.level.org.springframework=DEBUG`*

**logging.path**= *# Location of the log file. For instance `/var/log`*

**logging.pattern.console**= *# Appender pattern for output to the console. Only supported with the default logback setup.*

**logging.pattern.file**= *# Appender pattern for output to the file. Only supported with the default logback setup.*

**logging.pattern.level**= *# Appender pattern for log level (default %5p). Only supported with the default logback setup.*

**logging.register-shutdown-hook**=false *# Register a shutdown hook for the logging system when it is initialized.*

*# AOP*

**spring.aop.auto**=true *# Add @EnableAspectJAutoProxy.*

**spring.aop.proxy-target-class**= *# Whether subclass-based (CGLIB) proxies are to be created (true) as opposed to standard Java interface-based proxies (false). Defaults to "true" when using Spring Transaction Management, otherwise "false".*

*# IDENTITY (*[ContextIdApplicationContextInitializer](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/ContextIdApplicationContextInitializer.java))

**spring.application.index**= *# Application index.*

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

*# ADMIN (*[SpringApplicationAdminJmxAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/admin/SpringApplicationAdminJmxAutoConfiguration.java))

**spring.application.admin.enabled**=false *# Enable admin features for the application.*

**spring.application.admin.jmx-name**=org.springframework.boot:type=Admin,name=SpringApplication *# JMX name of the application admin MBean.*

*# AUTO-CONFIGURATION*

**spring.autoconfigure.exclude**= *# Auto-configuration classes to exclude.*

*# SPRING CORE*

**spring.beaninfo.ignore**=true *# Skip search of BeanInfo classes.*

*# SPRING CACHE (*[CacheProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cache/CacheProperties.java))

**spring.cache.cache-names**= *# Comma-separated list of cache names to create if supported by the underlying cache manager.*

**spring.cache.caffeine.spec**= *# The spec to use to create caches. Check CaffeineSpec for more details on the spec format.*

**spring.cache.couchbase.expiration**=0 *# Entry expiration in milliseconds. By default the entries never expire.*

**spring.cache.ehcache.config**= *# The location of the configuration file to use to initialize EhCache.*

**spring.cache.guava.spec**= *# The spec to use to create caches. Check CacheBuilderSpec for more details on the spec format.*

**spring.cache.infinispan.config**= *# The location of the configuration file to use to initialize Infinispan.*

**spring.cache.jcache.config**= *# The location of the configuration file to use to initialize the cache manager.*

**spring.cache.jcache.provider**= *# Fully qualified name of the CachingProvider implementation to use to retrieve the JSR-107 compliant cache manager. Only needed if more than one JSR-107 implementation is available on the classpath.*

**spring.cache.type**= *# Cache type, auto-detected according to the environment by default.*

*# SPRING CONFIG - using environment property only (*[ConfigFileApplicationListener](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/config/ConfigFileApplicationListener.java))

**spring.config.location**= *# Config file locations.*

**spring.config.name**=application *# Config file name.*

*# HAZELCAST (*[HazelcastProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hazelcast/HazelcastProperties.java))

**spring.hazelcast.config**= *# The location of the configuration file to use to initialize Hazelcast.*

*# PROJECT INFORMATION (*[ProjectInfoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/info/ProjectInfoProperties.java))

**spring.info.build.location**=classpath:META-INF/build-info.properties *# Location of the generated build- info.properties file.*

**spring.info.git.location**=classpath:git.properties *# Location of the generated git.properties file.*

*# JMX*

**spring.jmx.default-domain**= *# JMX domain name.* **spring.jmx.enabled**=true *# Expose management beans to the JMX domain.* **spring.jmx.server**=mbeanServer *# MBeanServer bean name.*

*# Email (*[MailProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mail/MailProperties.java))

**spring.mail.default-encoding**=UTF-8 *# Default MimeMessage encoding.*

**spring.mail.host**= *# SMTP server host. For instance `smtp.example.com`*

**spring.mail.jndi-name**= *# Session JNDI name. When set, takes precedence to others mail settings.*

**spring.mail.password**= *# Login password of the SMTP server.*

**spring.mail.port**= *# SMTP server port.*

**spring.mail.properties.\***= *# Additional JavaMail session properties.*

**spring.mail.protocol**=smtp *# Protocol used by the SMTP server.*

**spring.mail.test-connection**=false *# Test that the mail server is available on startup.*

**spring.mail.username**= *# Login user of the SMTP server.*

*# APPLICATION SETTINGS (*[SpringApplication](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/SpringApplication.java))

**spring.main.banner-mode**=console *# Mode used to display the banner when the application runs.*

**spring.main.sources**= *# Sources (class name, package name or XML resource location) to include in the ApplicationContext.*

**spring.main.web-environment**= *# Run the application in a web environment (auto-detected by default).*

*# FILE ENCODING (*[FileEncodingApplicationListener](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/FileEncodingApplicationListener.java))

**spring.mandatory-file-encoding**= *# Expected character encoding the application must use. # INTERNATIONALIZATION (*[MessageSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/context/MessageSourceAutoConfiguration.java))

**spring.messages.always-use-message-format**=false *# Set whether to always apply the MessageFormat rules, parsing even messages without arguments.*

**spring.messages.basename**=messages *# Comma-separated list of basenames, each following the ResourceBundle convention.*

**spring.messages.cache-seconds**=-1 *# Loaded resource bundle files cache expiration, in seconds. When set to -1, bundles are cached forever.*

**spring.messages.encoding**=UTF-8 *# Message bundles encoding.*

**spring.messages.fallback-to-system-locale**=true *# Set whether to fall back to the system Locale if no files for a specific Locale have been found.*

*# OUTPUT*

**spring.output.ansi.enabled**=detect *# Configure the ANSI output.*

*# PID FILE (*[ApplicationPidFileWriter](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/system/ApplicationPidFileWriter.java))

**spring.pid.fail-on-write-error**= *# Fail if ApplicationPidFileWriter is used but it cannot write the PID file.*

**spring.pid.file**= *# Location of the PID file to write (if ApplicationPidFileWriter is used).*

*# PROFILES*

**spring.profiles.active**= *# Comma-separated list (or list if using YAML) of* [active profiles](#_bookmark453).

**spring.profiles.include**= *# Unconditionally activate the specified comma separated profiles (or list of profiles if using YAML).*

*# SENDGRID (*[SendGridAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/sendgrid/SendGridAutoConfiguration.java))

**spring.sendgrid.api-key**= *# SendGrid api key (alternative to username/password).* **spring.sendgrid.username**= *# SendGrid account username.* **spring.sendgrid.password**= *# SendGrid account password.* **spring.sendgrid.proxy.host**= *# SendGrid proxy host.*

**spring.sendgrid.proxy.port**= *# SendGrid proxy port.*

*# ---------------------------------------- # WEB PROPERTIES*

*# ----------------------------------------*

*# EMBEDDED SERVER CONFIGURATION (*[ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java))

**server.address**= *# Network address to which the server should bind to.* **server.compression.enabled**=false *# If response compression is enabled.* **server.compression.excluded-user-agents**= *# List of user-agents to exclude from compression.* **server.compression.mime-types**=text/html,text/xml,text/plain,text/css,text/javascript,application/ javascript *# Comma-separated list of MIME types that should be compressed.*

**server.compression.min-response-size**=2048 *# Minimum response size that is required for compression to be performed.*

**server.connection-timeout**= *# Time in milliseconds that connectors will wait for another HTTP request before closing the connection. When not set, the connector's container-specific default will be used. Use a value of -1 to indicate no (i.e. infinite) timeout.*

**server.context-parameters.\***= *# Servlet context init parameters. For instance `server.context- parameters.a=alpha`*

**server.context-path**= *# Context path of the application.*

**server.display-name**=application *# Display name of the application.*

**server.max-http-header-size**=0 *# Maximum size in bytes of the HTTP message header.* **server.error.include-stacktrace**=never *# When to include a "stacktrace" attribute.* **server.error.path**=/error *# Path of the error controller.*

**server.error.whitelabel.enabled**=true *# Enable the default error page displayed in browsers in case of a server error.*

**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.*

**server.jsp-servlet.class-name**=org.apache.jasper.servlet.JspServlet *# The class name of the JSP servlet.*

**server.jsp-servlet.init-parameters.\***= *# Init parameters used to configure the JSP servlet* **server.jsp-servlet.registered**=true *# Whether or not the JSP servlet is registered* **server.port**=8080 *# Server HTTP port.*

**server.server-header**= *# Value to use for the Server response header (no header is sent if empty)*

**server.servlet-path**=/ *# Path of the main dispatcher servlet.*

**server.use-forward-headers**= *# If X-Forwarded-\* headers should be applied to the HttpRequest.* **server.session.cookie.comment**= *# Comment for the session cookie.* **server.session.cookie.domain**= *# Domain for the session cookie.*

**server.session.cookie.http-only**= *# "HttpOnly" flag for the session cookie.* **server.session.cookie.max-age**= *# Maximum age of the session cookie in seconds.* **server.session.cookie.name**= *# Session cookie name.*

**server.session.cookie.path**= *# Path of the session cookie.*

**server.session.cookie.secure**= *# "Secure" flag for the session cookie.* **server.session.persistent**=false *# Persist session data between restarts.* **server.session.store-dir**= *# Directory used to store session data.* **server.session.timeout**= *# Session timeout in seconds.*

**server.session.tracking-modes**= *# Session tracking modes (one or more of the following: "cookie", "url", "ssl").*

**server.ssl.ciphers**= *# Supported SSL ciphers.*

**server.ssl.client-auth**= *# Whether client authentication is wanted ("want") or needed ("need"). Requires a trust store.*

**server.ssl.enabled**= *# Enable SSL support.*

**server.ssl.enabled-protocols**= *# Enabled SSL protocols.*

**server.ssl.key-alias**= *# Alias that identifies the key in the key store.*

**server.ssl.key-password**= *# Password used to access the key in the key store.*

**server.ssl.key-store**= *# Path to the key store that holds the SSL certificate (typically a jks file).*

**server.ssl.key-store-password**= *# Password used to access the key store.* **server.ssl.key-store-provider**= *# Provider for the key store.* **server.ssl.key-store-type**= *# Type of the key store.* **server.ssl.protocol**=TLS *# SSL protocol to use.*

**server.ssl.trust-store**= *# Trust store that holds SSL certificates.* **server.ssl.trust-store-password**= *# Password used to access the trust store.* **server.ssl.trust-store-provider**= *# Provider for the trust store.* **server.ssl.trust-store-type**= *# Type of the trust store.*

**server.tomcat.accept-count**= *# Maximum queue length for incoming connection requests when all possible request processing threads are in use.*

**server.tomcat.accesslog.buffered**=true *# Buffer output such that it is only flushed periodically.*

**server.tomcat.accesslog.directory**=logs *# Directory in which log files are created. Can be relative to the tomcat base dir or absolute.*

**server.tomcat.accesslog.enabled**=false *# Enable access log.*

**server.tomcat.accesslog.file-date-format**=.yyyy-MM-dd *# Date format to place in log file name.* **server.tomcat.accesslog.pattern**=common *# Format pattern for access logs.* **server.tomcat.accesslog.prefix**=access\_log *# Log file name prefix.*

**server.tomcat.accesslog.rename-on-rotate**=false *# Defer inclusion of the date stamp in the file name until rotate time.*

**server.tomcat.accesslog.request-attributes-enabled**=false *# Set request attributes for IP address, Hostname, protocol and port used for the request.*

**server.tomcat.accesslog.rotate**=true *# Enable access log rotation.*

**server.tomcat.accesslog.suffix**=.log *# Log file name suffix.*

**server.tomcat.additional-tld-skip-patterns**= *# Comma-separated list of additional patterns that match jars to ignore for TLD scanning.*

**server.tomcat.background-processor-delay**=30 *# Delay in seconds between the invocation of backgroundProcess methods.*

**server.tomcat.basedir**= *# Tomcat base directory. If not specified a temporary directory will be used.*

**server.tomcat.internal-proxies**=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\ 192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\

169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\

127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\

172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\

172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\

172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3} *# regular expression matching trusted IP addresses.*

**server.tomcat.max-connections**= *# Maximum number of connections that the server will accept and process at any given time.*

**server.tomcat.max-http-post-size**=0 *# Maximum size in bytes of the HTTP post content.*

**server.tomcat.max-threads**=0 *# Maximum amount of worker threads.*

**server.tomcat.min-spare-threads**=0 *# Minimum amount of worker threads.*

**server.tomcat.port-header**=X-Forwarded-Port *# Name of the HTTP header used to override the original port value.*

**server.tomcat.protocol-header**= *# Header that holds the incoming protocol, usually named "X-Forwarded- Proto".*

**server.tomcat.protocol-header-https-value**=https *# Value of the protocol header that indicates that the incoming request uses SSL.*

**server.tomcat.redirect-context-root**= *# Whether requests to the context root should be redirected by appending a / to the path.*

**server.tomcat.remote-ip-header**= *# Name of the http header from which the remote ip is extracted. For instance `X-FORWARDED-FOR`*

**server.tomcat.uri-encoding**=UTF-8 *# Character encoding to use to decode the URI.* **server.undertow.accesslog.dir**= *# Undertow access log directory.* **server.undertow.accesslog.enabled**=false *# Enable access log.* **server.undertow.accesslog.pattern**=common *# Format pattern for access logs.* **server.undertow.accesslog.prefix**=access\_log. *# Log file name prefix.* **server.undertow.accesslog.rotate**=true *# Enable access log rotation.* **server.undertow.accesslog.suffix**=log *# Log file name suffix.*

**server.undertow.buffer-size**= *# Size of each buffer in bytes.* **server.undertow.direct-buffers**= *# Allocate buffers outside the Java heap.* **server.undertow.io-threads**= *# Number of I/O threads to create for the worker.*

**server.undertow.max-http-post-size**=0 *# Maximum size in bytes of the HTTP post content.*

**server.undertow.worker-threads**= *# Number of worker threads.*

*# FREEMARKER (*[FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java))

**spring.freemarker.allow-request-override**=false *# Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.*

**spring.freemarker.allow-session-override**=false *# Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.*

**spring.freemarker.cache**=false *# Enable template caching.*

**spring.freemarker.charset**=UTF-8 *# Template encoding.*

**spring.freemarker.check-template-location**=true *# Check that the templates location exists.* **spring.freemarker.content-type**=text/html *# Content-Type value.* **spring.freemarker.enabled**=true *# Enable MVC view resolution for this technology.*

**spring.freemarker.expose-request-attributes**=false *# Set whether all request attributes should be added to the model prior to merging with the template.*

**spring.freemarker.expose-session-attributes**=false *# Set whether all HttpSession attributes should be added to the model prior to merging with the template.*

**spring.freemarker.expose-spring-macro-helpers**=true *# Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".*

**spring.freemarker.prefer-file-system-access**=true *# Prefer file system access for template loading. File system access enables hot detection of template changes.*

**spring.freemarker.prefix**= *# Prefix that gets prepended to view names when building a URL.* **spring.freemarker.request-context-attribute**= *# Name of the RequestContext attribute for all views.* **spring.freemarker.settings.\***= *# Well-known FreeMarker keys which will be passed to FreeMarker's*

*Configuration.*

**spring.freemarker.suffix**=.ftl *# Suffix that gets appended to view names when building a URL.* **spring.freemarker.template-loader-path**=classpath:/templates/ *# Comma-separated list of template paths.* **spring.freemarker.view-names**= *# White list of view names that can be resolved.*

*# GROOVY TEMPLATES (*[GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java))

**spring.groovy.template.allow-request-override**=false *# Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.*

**spring.groovy.template.allow-session-override**=false *# Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.*

**spring.groovy.template.cache**= *# Enable template caching.*

**spring.groovy.template.charset**=UTF-8 *# Template encoding.*

**spring.groovy.template.check-template-location**=true *# Check that the templates location exists.* **spring.groovy.template.configuration.\***= *# See GroovyMarkupConfigurer* **spring.groovy.template.content-type**=test/html *# Content-Type value.* **spring.groovy.template.enabled**=true *# Enable MVC view resolution for this technology.*

**spring.groovy.template.expose-request-attributes**=false *# Set whether all request attributes should be added to the model prior to merging with the template.*

**spring.groovy.template.expose-session-attributes**=false *# Set whether all HttpSession attributes should be added to the model prior to merging with the template.*

**spring.groovy.template.expose-spring-macro-helpers**=true *# Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".*

**spring.groovy.template.prefix**= *# Prefix that gets prepended to view names when building a URL.* **spring.groovy.template.request-context-attribute**= *# Name of the RequestContext attribute for all views.* **spring.groovy.template.resource-loader-path**=classpath:/templates/ *# Template path.* **spring.groovy.template.suffix**=.tpl *# Suffix that gets appended to view names when building a URL.* **spring.groovy.template.view-names**= *# White list of view names that can be resolved.*

*# SPRING HATEOAS (*[HateoasProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hateoas/HateoasProperties.java))

**spring.hateoas.use-hal-as-default-json-media-type**=true *# Specify if application/hal+json responses should be sent to requests that accept application/json.*

*# HTTP message conversion*

**spring.http.converters.preferred-json-mapper**=jackson *# Preferred JSON mapper to use for HTTP message conversion. Set to "gson" to force the use of Gson when both it and Jackson are on the classpath.*

*# HTTP encoding (*[HttpEncodingProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/HttpEncodingProperties.java))

**spring.http.encoding.charset**=UTF-8 *# Charset of HTTP requests and responses. Added to the "Content-Type" header if not set explicitly.*

**spring.http.encoding.enabled**=true *# Enable http encoding support.*

**spring.http.encoding.force**= *# Force the encoding to the configured charset on HTTP requests and responses.*

**spring.http.encoding.force-request**= *# Force the encoding to the configured charset on HTTP requests. Defaults to true when "force" has not been specified.*

**spring.http.encoding.force-response**= *# Force the encoding to the configured charset on HTTP responses.*

**spring.http.encoding.mapping**= *# Locale to Encoding mapping.*

*# MULTIPART (*[MultipartProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/MultipartProperties.java))

**spring.http.multipart.enabled**=true *# Enable support of multi-part uploads.*

**spring.http.multipart.file-size-threshold**=0 *# Threshold after which files will be written to disk. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.*

**spring.http.multipart.location**= *# Intermediate location of uploaded files.*

**spring.http.multipart.max-file-size**=1MB *# Max file size. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.*

**spring.http.multipart.max-request-size**=10MB *# Max request size. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.*

**spring.http.multipart.resolve-lazily**=false *# Whether to resolve the multipart request lazily at the time of file or parameter access.*

*# JACKSON (*[JacksonProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jackson/JacksonProperties.java))

**spring.jackson.date-format**= *# Date format string or a fully-qualified date format class name. For instance `yyyy-MM-dd HH:mm:ss`.*

**spring.jackson.default-property-inclusion**= *# Controls the inclusion of properties during serialization.*

**spring.jackson.deserialization.\***= *# Jackson on/off features that affect the way Java objects are deserialized.*

**spring.jackson.generator.\***= *# Jackson on/off features for generators.*

**spring.jackson.joda-date-time-format**= *# Joda date time format string. If not configured, "date-format" will be used as a fallback if it is configured with a format string.*

**spring.jackson.locale**= *# Locale used for formatting.* **spring.jackson.mapper.\***= *# Jackson general purpose on/off features.* **spring.jackson.parser.\***= *# Jackson on/off features for parsers.*

**spring.jackson.property-naming-strategy**= *# One of the constants on Jackson's PropertyNamingStrategy. Can also be a fully-qualified class name of a PropertyNamingStrategy subclass.*

**spring.jackson.serialization.\***= *# Jackson on/off features that affect the way Java objects are serialized.*

**spring.jackson.time-zone**= *# Time zone used when formatting dates. For instance `America/Los\_Angeles`*

*# JERSEY (*[JerseyProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jersey/JerseyProperties.java))

**spring.jersey.application-path**= *# Path that serves as the base URI for the application. Overrides the value of "@ApplicationPath" if specified.*

**spring.jersey.filter.order**=0 *# Jersey filter chain order.*

**spring.jersey.init.\***= *# Init parameters to pass to Jersey via the servlet or filter.* **spring.jersey.servlet.load-on-startup**=-1 *# Load on startup priority of the Jersey servlet.* **spring.jersey.type**=servlet *# Jersey integration type.*

*# SPRING LDAP (*[LdapProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/ldap/LdapProperties.java))

**spring.ldap.urls**= *# LDAP URLs of the server.*

**spring.ldap.base**= *# Base suffix from which all operations should originate.*

**spring.ldap.username**= *# Login user of the server.* **spring.ldap.password**= *# Login password of the server.* **spring.ldap.base-environment.\***= *# LDAP specification settings.*

*# EMBEDDED LDAP (*[EmbeddedLdapProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/ldap/embedded/EmbeddedLdapProperties.java))

**spring.ldap.embedded.base-dn**= *# The base DN* **spring.ldap.embedded.credential.username**= *# Embedded LDAP username.* **spring.ldap.embedded.credential.password**= *# Embedded LDAP password.*

**spring.ldap.embedded.ldif**=classpath:schema.ldif *# Schema (LDIF) script resource reference.* **spring.ldap.embedded.port**= *# Embedded LDAP port.* **spring.ldap.embedded.validation.enabled**=true *# Enable LDAP schema validation.* **spring.ldap.embedded.validation.schema**= *# Path to the custom schema.*

*# SPRING MOBILE DEVICE VIEWS (*[DeviceDelegatingViewResolverAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/DeviceDelegatingViewResolverAutoConfiguration.java))

**spring.mobile.devicedelegatingviewresolver.enable-fallback**=false *# Enable support for fallback resolution.*

**spring.mobile.devicedelegatingviewresolver.enabled**=false *# Enable device view resolver.*

**spring.mobile.devicedelegatingviewresolver.mobile-prefix**=mobile/ *# Prefix that gets prepended to view names for mobile devices.*

**spring.mobile.devicedelegatingviewresolver.mobile-suffix**= *# Suffix that gets appended to view names for mobile devices.*

**spring.mobile.devicedelegatingviewresolver.normal-prefix**= *# Prefix that gets prepended to view names for normal devices.*

**spring.mobile.devicedelegatingviewresolver.normal-suffix**= *# Suffix that gets appended to view names for normal devices.*

**spring.mobile.devicedelegatingviewresolver.tablet-prefix**=tablet/ *# Prefix that gets prepended to view names for tablet devices.*

**spring.mobile.devicedelegatingviewresolver.tablet-suffix**= *# Suffix that gets appended to view names for tablet devices.*

*# SPRING MOBILE SITE PREFERENCE (*[SitePreferenceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/SitePreferenceAutoConfiguration.java))

**spring.mobile.sitepreference.enabled**=true *# Enable SitePreferenceHandler.*

*# MUSTACHE TEMPLATES (*[MustacheAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mustache/MustacheAutoConfiguration.java))

**spring.mustache.allow-request-override**= *# Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.*

**spring.mustache.allow-session-override**= *# Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.*

**spring.mustache.cache**= *# Enable template caching.*

**spring.mustache.charset**= *# Template encoding.*

**spring.mustache.check-template-location**= *# Check that the templates location exists.*

**spring.mustache.content-type**= *# Content-Type value.*

**spring.mustache.enabled**= *# Enable MVC view resolution for this technology.*

**spring.mustache.expose-request-attributes**= *# Set whether all request attributes should be added to the model prior to merging with the template.*

**spring.mustache.expose-session-attributes**= *# Set whether all HttpSession attributes should be added to the model prior to merging with the template.*

**spring.mustache.expose-spring-macro-helpers**= *# Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".*

**spring.mustache.prefix**=classpath:/templates/ *# Prefix to apply to template names.* **spring.mustache.request-context-attribute**= *# Name of the RequestContext attribute for all views.* **spring.mustache.suffix**=.html *# Suffix to apply to template names.*

**spring.mustache.view-names**= *# White list of view names that can be resolved.*

*# SPRING MVC (*[WebMvcProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcProperties.java))

**spring.mvc.async.request-timeout**= *# Amount of time (in milliseconds) before asynchronous request handling times out.*

**spring.mvc.date-format**= *# Date format to use. For instance `dd/MM/yyyy`.*

**spring.mvc.dispatch-trace-request**=false *# Dispatch TRACE requests to the FrameworkServlet doService method.*

**spring.mvc.dispatch-options-request**=true *# Dispatch OPTIONS requests to the FrameworkServlet doService method.*

**spring.mvc.favicon.enabled**=true *# Enable resolution of favicon.ico.* **spring.mvc.formcontent.putfilter.enabled**=true *# Enable Spring's HttpPutFormContentFilter.* **spring.mvc.ignore-default-model-on-redirect**=true *# If the content of the "default" model should be*

*ignored during redirect scenarios.*

**spring.mvc.locale**= *# Locale to use. By default, this locale is overridden by the "Accept-Language" header.*

**spring.mvc.locale-resolver**=accept-header *# Define how the locale should be resolved.*

**spring.mvc.log-resolved-exception**=false *# Enable warn logging of exceptions resolved by a "HandlerExceptionResolver".*

**spring.mvc.media-types.\***= *# Maps file extensions to media types for content negotiation.*

**spring.mvc.message-codes-resolver-format**= *# Formatting strategy for message codes. For instance*

*`PREFIX\_ERROR\_CODE`.*

**spring.mvc.servlet.load-on-startup**=-1 *# Load on startup priority of the Spring Web Services servlet.*

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

**spring.mvc.throw-exception-if-no-handler-found**=false *# If a "NoHandlerFoundException" should be thrown if no Handler was found to process a request.*

**spring.mvc.view.prefix**= *# Spring MVC view prefix.*

**spring.mvc.view.suffix**= *# Spring MVC view suffix.*

*# SPRING RESOURCES HANDLING (*[ResourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ResourceProperties.java))

**spring.resources.add-mappings**=true *# Enable default resource handling.*

**spring.resources.cache-period**= *# Cache period for the resources served by the resource handler, in seconds.*

**spring.resources.chain.cache**=true *# Enable caching in the Resource chain.*

**spring.resources.chain.enabled**= *# Enable the Spring Resource Handling chain. Disabled by default unless at least one strategy has been enabled.*

**spring.resources.chain.gzipped**=false *# Enable resolution of already gzipped resources.* **spring.resources.chain.html-application-cache**=false *# Enable HTML5 application cache manifest rewriting.* **spring.resources.chain.strategy.content.enabled**=false *# Enable the content Version Strategy.* **spring.resources.chain.strategy.content.paths**=/\*\* *# Comma-separated list of patterns to apply to the*

*Version Strategy.*

**spring.resources.chain.strategy.fixed.enabled**=false *# Enable the fixed Version Strategy.*

**spring.resources.chain.strategy.fixed.paths**=/\*\* *# Comma-separated list of patterns to apply to the Version Strategy.*

**spring.resources.chain.strategy.fixed.version**= *# Version string to use for the Version Strategy.*

**spring.resources.static-locations**=classpath:/META-INF/resources/,classpath:/resources/,classpath:/ static/,classpath:/public/ *# Locations of static resources.*

*# SPRING SESSION (*[SessionProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/session/SessionProperties.java))

**spring.session.hazelcast.flush-mode**=on-save *# Sessions flush mode.*

**spring.session.hazelcast.map-name**=spring:session:sessions *# Name of the map used to store sessions.*

**spring.session.jdbc.initializer.enabled**= *# Create the required session tables on startup if necessary. Enabled automatically if the default table name is set or a custom schema is configured.*

[**spring.session.jdbc.schema**=classpath:org/springframework/session/jdbc/schema-@@platform@@.sql](mailto:org/springframework/session/jdbc/schema-@@platform@@.sql) *# Path to the SQL file to use to initialize the database schema.*

**spring.session.jdbc.table-name**=SPRING\_SESSION *# Name of database table used to store sessions.* **spring.session.mongo.collection-name**=sessions *# Collection name used to store sessions.* **spring.session.redis.flush-mode**=on-save *# Sessions flush mode.*

**spring.session.redis.namespace**= *# Namespace for keys used to store sessions.*

**spring.session.store-type**= *# Session store type.*

*# SPRING SOCIAL (*[SocialWebAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/SocialWebAutoConfiguration.java))

**spring.social.auto-connection-views**=false *# Enable the connection status view for supported providers.*

*# SPRING SOCIAL FACEBOOK (*[FacebookAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/FacebookAutoConfiguration.java)) **spring.social.facebook.app-id**= *# your application's Facebook App ID* **spring.social.facebook.app-secret**= *# your application's Facebook App Secret*

*# SPRING SOCIAL LINKEDIN (*[LinkedInAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/LinkedInAutoConfiguration.java)) **spring.social.linkedin.app-id**= *# your application's LinkedIn App ID* **spring.social.linkedin.app-secret**= *# your application's LinkedIn App Secret*

*# SPRING SOCIAL TWITTER (*[TwitterAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/TwitterAutoConfiguration.java)) **spring.social.twitter.app-id**= *# your application's Twitter App ID* **spring.social.twitter.app-secret**= *# your application's Twitter App Secret*

*# THYMELEAF (*[ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java))

**spring.thymeleaf.cache**=true *# Enable template caching.*

**spring.thymeleaf.check-template**=true *# Check that the template exists before rendering it.* **spring.thymeleaf.check-template-location**=true *# Check that the templates location exists.* **spring.thymeleaf.content-type**=text/html *# Content-Type value.* **spring.thymeleaf.enabled**=true *# Enable MVC Thymeleaf view resolution.* **spring.thymeleaf.encoding**=UTF-8 *# Template encoding.*

**spring.thymeleaf.excluded-view-names**= *# Comma-separated list of view names that should be excluded from resolution.*

**spring.thymeleaf.mode**=HTML5 *# Template mode to be applied to templates. See also StandardTemplateModeHandlers.*

**spring.thymeleaf.prefix**=classpath:/templates/ *# Prefix that gets prepended to view names when building a URL.*

**spring.thymeleaf.suffix**=.html *# Suffix that gets appended to view names when building a URL.* **spring.thymeleaf.template-resolver-order**= *# Order of the template resolver in the chain.* **spring.thymeleaf.view-names**= *# Comma-separated list of view names that can be resolved.*

*# SPRING WEB SERVICES (*[WebServicesProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/webservices/WebServicesProperties.java))

**spring.webservices.path**=/services *# Path that serves as the base URI for the services.* **spring.webservices.servlet.init**= *# Servlet init parameters to pass to Spring Web Services.* **spring.webservices.servlet.load-on-startup**=-1 *# Load on startup priority of the Spring Web Services*

*servlet.*

*# ---------------------------------------- # SECURITY PROPERTIES*

*# ----------------------------------------*

*# SECURITY (*[SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java))

**security.basic.authorize-mode**=role *# Security authorize mode to apply.* **security.basic.enabled**=true *# Enable basic authentication.* **security.basic.path**=/\*\* *# Comma-separated list of paths to secure.* **security.basic.realm**=Spring *# HTTP basic realm name.*

**security.enable-csrf**=false *# Enable Cross Site Request Forgery support.*

**security.filter-order**=0 *# Security filter chain order.*

**security.filter-dispatcher-types**=ASYNC, FORWARD, INCLUDE, REQUEST *# Security filter chain dispatcher types.*

**security.headers.cache**=true *# Enable cache control HTTP headers.* **security.headers.content-security-policy**= *# Value for content security policy header.* **security.headers.content-security-policy-mode**=default *# Content security policy mode.*

**security.headers.content-type**=true *# Enable "X-Content-Type-Options" header.*

**security.headers.frame**=true *# Enable "X-Frame-Options" header.*

**security.headers.hsts**=all *# HTTP Strict Transport Security (HSTS) mode (none, domain, all).*

**security.headers.xss**=true *# Enable cross site scripting (XSS) protection.*

**security.ignored**= *# Comma-separated list of paths to exclude from the default secured paths.* **security.require-ssl**=false *# Enable secure channel for all requests.* **security.sessions**=stateless *# Session creation policy (always, never, if\_required, stateless).* **security.user.name**=user *# Default user name.*

**security.user.password**= *# Password for the default user name. A random password is logged on startup by default.*

**security.user.role**=USER *# Granted roles for the default user name.*

*# SECURITY OAUTH2 CLIENT (*[OAuth2ClientProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/OAuth2ClientProperties.java))

**security.oauth2.client.client-id**= *# OAuth2 client id.*

**security.oauth2.client.client-secret**= *# OAuth2 client secret. A random secret is generated by default*

*# SECURITY OAUTH2 RESOURCES (*[ResourceServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/resource/ResourceServerProperties.java))

**security.oauth2.resource.filter-order**= *# The order of the filter chain used to authenticate tokens.*

**security.oauth2.resource.id**= *# Identifier of the resource.*

**security.oauth2.resource.jwt.key-uri**= *# The URI of the JWT token. Can be set if the value is not available and the key is public.*

**security.oauth2.resource.jwt.key-value**= *# The verification key of the JWT token. Can either be a symmetric secret or PEM-encoded RSA public key.*

**security.oauth2.resource.jwk.key-set-uri**= *# The URI for getting the set of keys that can be used to validate the token.*

**security.oauth2.resource.prefer-token-info**=true *# Use the token info, can be set to false to use the user info.*

**security.oauth2.resource.service-id**=resource *#*

**security.oauth2.resource.token-info-uri**= *# URI of the token decoding endpoint.* **security.oauth2.resource.token-type**= *# The token type to send when using the userInfoUri.* **security.oauth2.resource.user-info-uri**= *# URI of the user endpoint.*

*# SECURITY OAUTH2 SSO (*[OAuth2SsoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/client/OAuth2SsoProperties.java))

**security.oauth2.sso.filter-order**= *# Filter order to apply if not providing an explicit WebSecurityConfigurerAdapter*

**security.oauth2.sso.login-path**=/login *# Path to the login page, i.e. the one that triggers the redirect to the OAuth2 Authorization Server*

*# ---------------------------------------- # DATA PROPERTIES*

*# ----------------------------------------*

*# FLYWAY (*[FlywayProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayProperties.java))

**flyway.baseline-description**= *#*

**flyway.baseline-version**=1 *# version to start migration*

**flyway.baseline-on-migrate**= *#*

**flyway.check-location**=false *# Check that migration scripts location exists.*

**flyway.clean-on-validation-error**= *#* **flyway.enabled**=true *# Enable flyway.* **flyway.encoding**= *#*

**flyway.ignore-failed-future-migration**= *#*

**flyway.init-sqls**= *# SQL statements to execute to initialize a connection immediately after obtaining it.*

**flyway.locations**=classpath:db/migration *# locations of migrations scripts*

**flyway.out-of-order**= *#*

**flyway.password**= *# JDBC password if you want Flyway to create its own DataSource*

**flyway.placeholder-prefix**= *#* **flyway.placeholder-replacement**= *#* **flyway.placeholder-suffix**= *#* **flyway.placeholders.\***= *#* **flyway.schemas**= *# schemas to update* **flyway.sql-migration-prefix**=V *#* **flyway.sql-migration-separator**= *#* **flyway.sql-migration-suffix**=.sql *#* **flyway.table**= *#*

**flyway.url**= *# JDBC url of the database to migrate. If not set, the primary configured data source is used.*

**flyway.user**= *# Login user of the database to migrate.*

**flyway.validate-on-migrate**= *#*

*# LIQUIBASE (*[LiquibaseProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/liquibase/LiquibaseProperties.java))

**liquibase.change-log**=classpath:/db/changelog/db.changelog-master.yaml *# Change log configuration path.*

**liquibase.check-change-log-location**=true *# Check the change log location exists.* **liquibase.contexts**= *# Comma-separated list of runtime contexts to use.* **liquibase.default-schema**= *# Default database schema.*

**liquibase.drop-first**=false *# Drop the database schema first.* **liquibase.enabled**=true *# Enable liquibase support.* **liquibase.labels**= *# Comma-separated list of runtime labels to use.* **liquibase.parameters.\***= *# Change log parameters.* **liquibase.password**= *# Login password of the database to migrate.*

**liquibase.rollback-file**= *# File to which rollback SQL will be written when an update is performed.*

**liquibase.url**= *# JDBC url of the database to migrate. If not set, the primary configured data source is used.*

**liquibase.user**= *# Login user of the database to migrate.*

*# COUCHBASE (*[CouchbaseProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/couchbase/CouchbaseProperties.java))

**spring.couchbase.bootstrap-hosts**= *# Couchbase nodes (host or IP address) to bootstrap from.* **spring.couchbase.bucket.name**=default *# Name of the bucket to connect to.* **spring.couchbase.bucket.password**= *# Password of the bucket.*

**spring.couchbase.env.endpoints.key-value**=1 *# Number of sockets per node against the Key/value service.* **spring.couchbase.env.endpoints.query**=1 *# Number of sockets per node against the Query (N1QL) service.* **spring.couchbase.env.endpoints.view**=1 *# Number of sockets per node against the view service.* **spring.couchbase.env.ssl.enabled**= *# Enable SSL support. Enabled automatically if a "keyStore" is*

*provided unless specified otherwise.*

**spring.couchbase.env.ssl.key-store**= *# Path to the JVM key store that holds the certificates.* **spring.couchbase.env.ssl.key-store-password**= *# Password used to access the key store.* **spring.couchbase.env.timeouts.connect**=5000 *# Bucket connections timeout in milliseconds.* **spring.couchbase.env.timeouts.key-value**=2500 *# Blocking operations performed on a specific key timeout*

*in milliseconds.*

**spring.couchbase.env.timeouts.query**=7500 *# N1QL query operations timeout in milliseconds.* **spring.couchbase.env.timeouts.socket-connect**=1000 *# Socket connect connections timeout in milliseconds.* **spring.couchbase.env.timeouts.view**=7500 *# Regular and geospatial view operations timeout in*

*milliseconds.*

*# DAO (*[PersistenceExceptionTranslationAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/dao/PersistenceExceptionTranslationAutoConfiguration.java))

**spring.dao.exceptiontranslation.enabled**=true *# Enable the PersistenceExceptionTranslationPostProcessor.*

*# CASSANDRA (*[CassandraProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cassandra/CassandraProperties.java))

**spring.data.cassandra.cluster-name**= *# Name of the Cassandra cluster.* **spring.data.cassandra.compression**=none *# Compression supported by the Cassandra binary protocol.* **spring.data.cassandra.connect-timeout-millis**= *# Socket option: connection time out.* **spring.data.cassandra.consistency-level**= *# Queries consistency level.* **spring.data.cassandra.contact-points**=localhost *# Comma-separated list of cluster node addresses.* **spring.data.cassandra.fetch-size**= *# Queries default fetch size.*

**spring.data.cassandra.keyspace-name**= *# Keyspace name to use.* **spring.data.cassandra.load-balancing-policy**= *# Class name of the load balancing policy.* **spring.data.cassandra.port**= *# Port of the Cassandra server.* **spring.data.cassandra.password**= *# Login password of the server.* **spring.data.cassandra.read-timeout-millis**= *# Socket option: read time out.* **spring.data.cassandra.reconnection-policy**= *# Reconnection policy class.* **spring.data.cassandra.repositories.enabled**= *# Enable Cassandra repositories.* **spring.data.cassandra.retry-policy**= *# Class name of the retry policy.* **spring.data.cassandra.serial-consistency-level**= *# Queries serial consistency level.* **spring.data.cassandra.schema-action**=none *# Schema action to take at startup.* **spring.data.cassandra.ssl**=false *# Enable SSL support.*

**spring.data.cassandra.username**= *# Login user of the server.*

*# DATA COUCHBASE (*[CouchbaseDataProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/couchbase/CouchbaseDataProperties.java))

**spring.data.couchbase.auto-index**=false *# Automatically create views and indexes.*

**spring.data.couchbase.consistency**=read-your-own-writes *# Consistency to apply by default on generated queries.*

**spring.data.couchbase.repositories.enabled**=true *# Enable Couchbase repositories.*

*# ELASTICSEARCH (*[ElasticsearchProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchProperties.java))

**spring.data.elasticsearch.cluster-name**=elasticsearch *# Elasticsearch cluster name.*

**spring.data.elasticsearch.cluster-nodes**= *# Comma-separated list of cluster node addresses. If not specified, starts a client node.*

**spring.data.elasticsearch.properties.\***= *# Additional properties used to configure the client.*

**spring.data.elasticsearch.repositories.enabled**=true *# Enable Elasticsearch repositories.*

*# DATA LDAP*

**spring.data.ldap.repositories.enabled**=true *# Enable LDAP repositories.*

*# MONGODB (*[MongoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mongo/MongoProperties.java))

**spring.data.mongodb.authentication-database**= *# Authentication database name.*

**spring.data.mongodb.database**=test *# Database name.*

**spring.data.mongodb.field-naming-strategy**= *# Fully qualified name of the FieldNamingStrategy to use.*

**spring.data.mongodb.grid-fs-database**= *# GridFS database name.* **spring.data.mongodb.host**=localhost *# Mongo server host. Cannot be set with uri.* **spring.data.mongodb.password**= *# Login password of the mongo server. Cannot be set with uri.* **spring.data.mongodb.port**=27017 *# Mongo server port. Cannot be set with uri.* **spring.data.mongodb.repositories.enabled**=true *# Enable Mongo repositories.*

**spring.data.mongodb.uri**=mongodb://localhost/test *# Mongo database URI. Cannot be set with host, port and credentials.*

**spring.data.mongodb.username**= *# Login user of the mongo server. Cannot be set with uri.*

*# DATA REDIS*

**spring.data.redis.repositories.enabled**=true *# Enable Redis repositories.*

*# NEO4J (*[Neo4jProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/neo4j/Neo4jProperties.java))

**spring.data.neo4j.compiler**= *# Compiler to use.*

**spring.data.neo4j.embedded.enabled**=true *# Enable embedded mode if the embedded driver is available.*

**spring.data.neo4j.open-in-view**=false *# Register OpenSessionInViewInterceptor. Binds a Neo4j Session to the thread for the entire processing of the request.*

**spring.data.neo4j.password**= *# Login password of the server.* **spring.data.neo4j.repositories.enabled**=true *# Enable Neo4j repositories.* **spring.data.neo4j.uri**= *# URI used by the driver. Auto-detected by default.* **spring.data.neo4j.username**= *# Login user of the server.*

*# DATA REST (*[RepositoryRestProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/rest/RepositoryRestProperties.java))

**spring.data.rest.base-path**= *# Base path to be used by Spring Data REST to expose repository resources.*

**spring.data.rest.default-page-size**= *# Default size of pages.*

**spring.data.rest.detection-strategy**=default *# Strategy to use to determine which repositories get exposed.*

**spring.data.rest.enable-enum-translation**= *# Enable enum value translation via the Spring Data REST default resource bundle.*

**spring.data.rest.limit-param-name**= *# Name of the URL query string parameter that indicates how many results to return at once.*

**spring.data.rest.max-page-size**= *# Maximum size of pages.*

**spring.data.rest.page-param-name**= *# Name of the URL query string parameter that indicates what page to return.*

**spring.data.rest.return-body-on-create**= *# Return a response body after creating an entity.* **spring.data.rest.return-body-on-update**= *# Return a response body after updating an entity.* **spring.data.rest.sort-param-name**= *# Name of the URL query string parameter that indicates what direction*

*to sort results.*

*# SOLR (*[SolrProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/solr/SolrProperties.java))

**spring.data.solr.host**=http://127.0.0.1:8983/solr *# Solr host. Ignored if "zk-host" is set.* **spring.data.solr.repositories.enabled**=true *# Enable Solr repositories.* **spring.data.solr.zk-host**= *# ZooKeeper host address in the form HOST:PORT.*

*# DATASOURCE (*[DataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.java) & [DataSourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceProperties.java))

**spring.datasource.continue-on-error**=false *# Do not stop if an error occurs while initializing the database.*

**spring.datasource.data**= *# Data (DML) script resource references.*

**spring.datasource.data-username**= *# User of the database to execute DML scripts (if different).* **spring.datasource.data-password**= *# Password of the database to execute DML scripts (if different).* **spring.datasource.dbcp2.\***= *# Commons DBCP2 specific settings*

**spring.datasource.driver-class-name**= *# Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.*

**spring.datasource.generate-unique-name**=false *# Generate a random datasource name.* **spring.datasource.hikari.\***= *# Hikari specific settings* **spring.datasource.initialize**=true *# Populate the database using 'data.sql'.*

**spring.datasource.jmx-enabled**=false *# Enable JMX support (if provided by the underlying pool).*

**spring.datasource.jndi-name**= *# JNDI location of the datasource. Class, url, username & password are ignored when set.*

**spring.datasource.name**=testdb *# Name of the datasource.*

**spring.datasource.password**= *# Login password of the database.*

**spring.datasource.platform**=all *# Platform to use in the DDL or DML scripts (e.g. schema-${platform}.sql or data-${platform}.sql).*

**spring.datasource.schema**= *# Schema (DDL) script resource references.*

**spring.datasource.schema-username**= *# User of the database to execute DDL scripts (if different).*

**spring.datasource.schema-password**= *# Password of the database to execute DDL scripts (if different).*

**spring.datasource.separator**=; *# Statement separator in SQL initialization scripts.* **spring.datasource.sql-script-encoding**= *# SQL scripts encoding.* **spring.datasource.tomcat.\***= *# Tomcat datasource specific settings*

**spring.datasource.type**= *# Fully qualified name of the connection pool implementation to use. By default, it is auto-detected from the classpath.*

**spring.datasource.url**= *# JDBC url of the database.*

**spring.datasource.username**= *# Login user of the database.* **spring.datasource.xa.data-source-class-name**= *# XA datasource fully qualified name.* **spring.datasource.xa.properties**= *# Properties to pass to the XA data source.*

*# JEST (Elasticsearch HTTP client) (*[JestProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/elasticsearch/jest/JestProperties.java))

**spring.elasticsearch.jest.connection-timeout**=3000 *# Connection timeout in milliseconds.*

**spring.elasticsearch.jest.multi-threaded**=true *# Enable connection requests from multiple execution threads.*

**spring.elasticsearch.jest.password**= *# Login password.* **spring.elasticsearch.jest.proxy.host**= *# Proxy host the HTTP client should use.* **spring.elasticsearch.jest.proxy.port**= *# Proxy port the HTTP client should use.* **spring.elasticsearch.jest.read-timeout**=3000 *# Read timeout in milliseconds.*

**spring.elasticsearch.jest.uris**=http://localhost:9200 *# Comma-separated list of the Elasticsearch instances to use.*

**spring.elasticsearch.jest.username**= *# Login user.*

*# H2 Web Console (*[H2ConsoleProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/h2/H2ConsoleProperties.java))

**spring.h2.console.enabled**=false *# Enable the console.*

**spring.h2.console.path**=/h2-console *# Path at which the console will be available.* **spring.h2.console.settings.trace**=false *# Enable trace output.* **spring.h2.console.settings.web-allow-others**=false *# Enable remote access.*

*# JOOQ (*[JooqAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jooq/JooqAutoConfiguration.java))

**spring.jooq.sql-dialect**= *# SQLDialect JOOQ used when communicating with the configured datasource. For instance `POSTGRES`*

*# JPA (*[JpaBaseConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/JpaBaseConfiguration.java), [HibernateJpaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.java))

**spring.data.jpa.repositories.enabled**=true *# Enable JPA repositories.*

**spring.jpa.database**= *# Target database to operate on, auto-detected by default. Can be alternatively set using the "databasePlatform" property.*

**spring.jpa.database-platform**= *# Name of the target database to operate on, auto-detected by default. Can be alternatively set using the "Database" enum.*

**spring.jpa.generate-ddl**=false *# Initialize the schema on startup.*

**spring.jpa.hibernate.ddl-auto**= *# DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto" property. Default to "create-drop" when using an embedded database, "none" otherwise.*

**spring.jpa.hibernate.naming.implicit-strategy**= *# Hibernate 5 implicit naming strategy fully qualified name.*

**spring.jpa.hibernate.naming.physical-strategy**= *# Hibernate 5 physical naming strategy fully qualified name.*

**spring.jpa.hibernate.naming.strategy**= *# Hibernate 4 naming strategy fully qualified name. Not supported with Hibernate 5.*

**spring.jpa.hibernate.use-new-id-generator-mappings**= *# Use Hibernate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE.*

**spring.jpa.open-in-view**=true *# Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to the thread for the entire processing of the request.*

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

**spring.jpa.show-sql**=false *# Enable logging of SQL statements.*

*# JTA (*[JtaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/transaction/jta/JtaAutoConfiguration.java)) **spring.jta.enabled**=true *# Enable JTA support.* **spring.jta.log-dir**= *# Transaction logs directory.*

**spring.jta.transaction-manager-id**= *# Transaction manager unique identifier.*

*# ATOMIKOS (*[AtomikosProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/atomikos/AtomikosProperties.java))

**spring.jta.atomikos.connectionfactory.borrow-connection-timeout**=30 *# Timeout, in seconds, for borrowing connections from the pool.*

**spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag**=true *# Whether or not to ignore the transacted flag when creating session.*

**spring.jta.atomikos.connectionfactory.local-transaction-mode**=false *# Whether or not local transactions are desired.*

**spring.jta.atomikos.connectionfactory.maintenance-interval**=60 *# The time, in seconds, between runs of the pool's maintenance thread.*

**spring.jta.atomikos.connectionfactory.max-idle-time**=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

**spring.jta.atomikos.connectionfactory.max-lifetime**=0 *# The time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.*

**spring.jta.atomikos.connectionfactory.max-pool-size**=1 *# The maximum size of the pool.* **spring.jta.atomikos.connectionfactory.min-pool-size**=1 *# The minimum size of the pool.* **spring.jta.atomikos.connectionfactory.reap-timeout**=0 *# The reap timeout, in seconds, for borrowed*

*connections. 0 denotes no limit.*

**spring.jta.atomikos.connectionfactory.unique-resource-name**=jmsConnectionFactory *# The unique name used to identify the resource during recovery.*

**spring.jta.atomikos.datasource.borrow-connection-timeout**=30 *# Timeout, in seconds, for borrowing connections from the pool.*

**spring.jta.atomikos.datasource.default-isolation-level**= *# Default isolation level of connections provided by the pool.*

**spring.jta.atomikos.datasource.login-timeout**= *# Timeout, in seconds, for establishing a database connection.*

**spring.jta.atomikos.datasource.maintenance-interval**=60 *# The time, in seconds, between runs of the pool's maintenance thread.*

**spring.jta.atomikos.datasource.max-idle-time**=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

**spring.jta.atomikos.datasource.max-lifetime**=0 *# The time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.*

**spring.jta.atomikos.datasource.max-pool-size**=1 *# The maximum size of the pool.* **spring.jta.atomikos.datasource.min-pool-size**=1 *# The minimum size of the pool.* **spring.jta.atomikos.datasource.reap-timeout**=0 *# The reap timeout, in seconds, for borrowed connections.*

*0 denotes no limit.*

**spring.jta.atomikos.datasource.test-query**= *# SQL query or statement used to validate a connection before returning it.*

**spring.jta.atomikos.datasource.unique-resource-name**=dataSource *# The unique name used to identify the resource during recovery.*

**spring.jta.atomikos.properties.checkpoint-interval**=500 *# Interval between checkpoints.* **spring.jta.atomikos.properties.default-jta-timeout**=10000 *# Default timeout for JTA transactions.* **spring.jta.atomikos.properties.enable-logging**=true *# Enable disk logging.* **spring.jta.atomikos.properties.force-shutdown-on-vm-exit**=false *# Specify if a VM shutdown should trigger*

*forced shutdown of the transaction core.*

**spring.jta.atomikos.properties.log-base-dir**= *# Directory in which the log files should be stored.* **spring.jta.atomikos.properties.log-base-name**=tmlog *# Transactions log file base name.* **spring.jta.atomikos.properties.max-actives**=50 *# Maximum number of active transactions.* **spring.jta.atomikos.properties.max-timeout**=300000 *# Maximum timeout (in milliseconds) that can be*

*allowed for transactions.*

**spring.jta.atomikos.properties.serial-jta-transactions**=true *# Specify if sub-transactions should be joined when possible.*

**spring.jta.atomikos.properties.service**= *# Transaction manager implementation that should be started.*

**spring.jta.atomikos.properties.threaded-two-phase-commit**=false *# Use different (and concurrent) threads for two-phase commit on the participating resources.*

**spring.jta.atomikos.properties.transaction-manager-unique-name**= *# Transaction manager's unique name.*

*# BITRONIX*

**spring.jta.bitronix.connectionfactory.acquire-increment**=1 *# Number of connections to create when growing the pool.*

**spring.jta.bitronix.connectionfactory.acquisition-interval**=1 *# Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.*

**spring.jta.bitronix.connectionfactory.acquisition-timeout**=30 *# Timeout, in seconds, for acquiring connections from the pool.*

**spring.jta.bitronix.connectionfactory.allow-local-transactions**=true *# Whether or not the transaction manager should allow mixing XA and non-XA transactions.*

**spring.jta.bitronix.connectionfactory.apply-transaction-timeout**=false *# Whether or not the transaction timeout should be set on the XAResource when it is enlisted.*

**spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled**=true *# Whether or not resources should be enlisted and delisted automatically.*

**spring.jta.bitronix.connectionfactory.cache-producers-consumers**=true *# Whether or not produces and consumers should be cached.*

**spring.jta.bitronix.connectionfactory.defer-connection-release**=true *# Whether or not the provider can run many transactions on the same connection and supports transaction interleaving.*

**spring.jta.bitronix.connectionfactory.ignore-recovery-failures**=false *# Whether or not recovery failures should be ignored.*

**spring.jta.bitronix.connectionfactory.max-idle-time**=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

**spring.jta.bitronix.connectionfactory.max-pool-size**=10 *# The maximum size of the pool. 0 denotes no limit.*

**spring.jta.bitronix.connectionfactory.min-pool-size**=0 *# The minimum size of the pool.*

**spring.jta.bitronix.connectionfactory.password**= *# The password to use to connect to the JMS provider.*

**spring.jta.bitronix.connectionfactory.share-transaction-connections**=false *# Whether or not connections in the ACCESSIBLE state can be shared within the context of a transaction.*

**spring.jta.bitronix.connectionfactory.test-connections**=true *# Whether or not connections should be tested when acquired from the pool.*

**spring.jta.bitronix.connectionfactory.two-pc-ordering-position**=1 *# The position that this resource should take during two-phase commit (always first is Integer.MIN\_VALUE, always last is Integer.MAX\_VALUE).*

**spring.jta.bitronix.connectionfactory.unique-name**=jmsConnectionFactory *# The unique name used to identify the resource during recovery.*

**spring.jta.bitronix.connectionfactory.use-tm-join**=true Whether or not TMJOIN should be used when starting XAResources.

**spring.jta.bitronix.connectionfactory.user**= *# The user to use to connect to the JMS provider.*

**spring.jta.bitronix.datasource.acquire-increment**=1 *# Number of connections to create when growing the pool.*

**spring.jta.bitronix.datasource.acquisition-interval**=1 *# Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.*

**spring.jta.bitronix.datasource.acquisition-timeout**=30 *# Timeout, in seconds, for acquiring connections from the pool.*

**spring.jta.bitronix.datasource.allow-local-transactions**=true *# Whether or not the transaction manager should allow mixing XA and non-XA transactions.*

**spring.jta.bitronix.datasource.apply-transaction-timeout**=false *# Whether or not the transaction timeout should be set on the XAResource when it is enlisted.*

**spring.jta.bitronix.datasource.automatic-enlisting-enabled**=true *# Whether or not resources should be enlisted and delisted automatically.*

**spring.jta.bitronix.datasource.cursor-holdability**= *# The default cursor holdability for connections.*

**spring.jta.bitronix.datasource.defer-connection-release**=true *# Whether or not the database can run many transactions on the same connection and supports transaction interleaving.*

**spring.jta.bitronix.datasource.enable-jdbc4-connection-test**= *# Whether or not Connection.isValid() is called when acquiring a connection from the pool.*

**spring.jta.bitronix.datasource.ignore-recovery-failures**=false *# Whether or not recovery failures should be ignored.*

**spring.jta.bitronix.datasource.isolation-level**= *# The default isolation level for connections.* **spring.jta.bitronix.datasource.local-auto-commit**= *# The default auto-commit mode for local transactions.* **spring.jta.bitronix.datasource.login-timeout**= *# Timeout, in seconds, for establishing a database*

*connection.*

**spring.jta.bitronix.datasource.max-idle-time**=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

**spring.jta.bitronix.datasource.max-pool-size**=10 *# The maximum size of the pool. 0 denotes no limit.* **spring.jta.bitronix.datasource.min-pool-size**=0 *# The minimum size of the pool.* **spring.jta.bitronix.datasource.prepared-statement-cache-size**=0 *# The target size of the prepared*

*statement cache. 0 disables the cache.*

**spring.jta.bitronix.datasource.share-transaction-connections**=false *# Whether or not connections in the ACCESSIBLE state can be shared within the context of a transaction.*

**spring.jta.bitronix.datasource.test-query**= *# SQL query or statement used to validate a connection before returning it.*

**spring.jta.bitronix.datasource.two-pc-ordering-position**=1 *# The position that this resource should take during two-phase commit (always first is Integer.MIN\_VALUE, always last is Integer.MAX\_VALUE).*

**spring.jta.bitronix.datasource.unique-name**=dataSource *# The unique name used to identify the resource during recovery.*

**spring.jta.bitronix.datasource.use-tm-join**=true Whether or not TMJOIN should be used when starting XAResources.

**spring.jta.bitronix.properties.allow-multiple-lrc**=false *# Allow multiple LRC resources to be enlisted into the same transaction.*

**spring.jta.bitronix.properties.asynchronous2-pc**=false *# Enable asynchronously execution of two phase commit.*

**spring.jta.bitronix.properties.background-recovery-interval-seconds**=60 *# Interval in seconds at which to run the recovery process in the background.*

**spring.jta.bitronix.properties.current-node-only-recovery**=true *# Recover only the current node.*

**spring.jta.bitronix.properties.debug-zero-resource-transaction**=false *# Log the creation and commit call stacks of transactions executed without a single enlisted resource.*

**spring.jta.bitronix.properties.default-transaction-timeout**=60 *# Default transaction timeout in seconds.* **spring.jta.bitronix.properties.disable-jmx**=false *# Enable JMX support.* **spring.jta.bitronix.properties.exception-analyzer**= *# Set the fully qualified name of the exception*

*analyzer implementation to use.*

**spring.jta.bitronix.properties.filter-log-status**=false *# Enable filtering of logs so that only mandatory logs are written.*

**spring.jta.bitronix.properties.force-batching-enabled**=true *# Set if disk forces are batched.* **spring.jta.bitronix.properties.forced-write-enabled**=true *# Set if logs are forced to disk.* **spring.jta.bitronix.properties.graceful-shutdown-interval**=60 *# Maximum amount of seconds the TM will*

*wait for transactions to get done before aborting them at shutdown time.*

**spring.jta.bitronix.properties.jndi-transaction-synchronization-registry-name**= *# JNDI name of the TransactionSynchronizationRegistry.*

**spring.jta.bitronix.properties.jndi-user-transaction-name**= *# JNDI name of the UserTransaction.*

**spring.jta.bitronix.properties.journal**=disk *# Name of the journal. Can be 'disk', 'null' or a class name.*

**spring.jta.bitronix.properties.log-part1-filename**=btm1.tlog *# Name of the first fragment of the journal.*

**spring.jta.bitronix.properties.log-part2-filename**=btm2.tlog *# Name of the second fragment of the journal.*

**spring.jta.bitronix.properties.max-log-size-in-mb**=2 *# Maximum size in megabytes of the journal fragments.*

**spring.jta.bitronix.properties.resource-configuration-filename**= *# ResourceLoader configuration file name.*

**spring.jta.bitronix.properties.server-id**= *# ASCII ID that must uniquely identify this TM instance. Default to the machine's IP address.*

**spring.jta.bitronix.properties.skip-corrupted-logs**=false *# Skip corrupted transactions log entries.*

**spring.jta.bitronix.properties.warn-about-zero-resource-transaction**=true *# Log a warning for transactions executed without a single enlisted resource.*

*# NARAYANA (*[NarayanaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/narayana/NarayanaProperties.java))

**spring.jta.narayana.default-timeout**=60 *# Transaction timeout in seconds.* **spring.jta.narayana.expiry- scanners**=com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner *# Comma- separated list of expiry scanners.*

**spring.jta.narayana.log-dir**= *# Transaction object store directory.*

**spring.jta.narayana.one-phase-commit**=true *# Enable one phase commit optimisation.*

**spring.jta.narayana.periodic-recovery-period**=120 *# Interval in which periodic recovery scans are performed in seconds.*

**spring.jta.narayana.recovery-backoff-period**=10 *# Back off period between first and second phases of the recovery scan in seconds.*

**spring.jta.narayana.recovery-db-pass**= *# Database password to be used by recovery manager.* **spring.jta.narayana.recovery-db-user**= *# Database username to be used by recovery manager.* **spring.jta.narayana.recovery-jms-pass**= *# JMS password to be used by recovery manager.* **spring.jta.narayana.recovery-jms-user**= *# JMS username to be used by recovery manager.* **spring.jta.narayana.recovery-modules**= *# Comma-separated list of recovery modules.* **spring.jta.narayana.transaction-manager-id**=1 *# Unique transaction manager id.* **spring.jta.narayana.xa-resource-orphan-filters**= *# Comma-separated list of orphan filters.*

*# EMBEDDED MONGODB (*[EmbeddedMongoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mongo/embedded/EmbeddedMongoProperties.java))

**spring.mongodb.embedded.features**=SYNC\_DELAY *# Comma-separated list of features to enable.* **spring.mongodb.embedded.storage.database-dir**= *# Directory used for data storage.* **spring.mongodb.embedded.storage.oplog-size**= *# Maximum size of the oplog in megabytes.* **spring.mongodb.embedded.storage.repl-set-name**= *# Name of the replica set.* **spring.mongodb.embedded.version**=2.6.10 *# Version of Mongo to use.*

*# REDIS (*[RedisProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/redis/RedisProperties.java))

**spring.redis.cluster.max-redirects**= *# Maximum number of redirects to follow when executing commands across the cluster.*

**spring.redis.cluster.nodes**= *# Comma-separated list of "host:port" pairs to bootstrap from.*

**spring.redis.database**=0 *# Database index used by the connection factory.*

**spring.redis.url**= *# Connection URL, will override host, port and password (user will be ignored), e.g. redis://user:password@example.com:6379*

**spring.redis.host**=localhost *# Redis server host.* **spring.redis.password**= *# Login password of the redis server.* **spring.redis.ssl**=false *# Enable SSL support.*

**spring.redis.pool.max-active**=8 *# Max number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.*

**spring.redis.pool.max-idle**=8 *# Max number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections.*

**spring.redis.pool.max-wait**=-1 *# Maximum amount of time (in milliseconds) a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.*

**spring.redis.pool.min-idle**=0 *# Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if it is positive.*

**spring.redis.port**=6379 *# Redis server port.* **spring.redis.sentinel.master**= *# Name of Redis server.* **spring.redis.sentinel.nodes**= *# Comma-separated list of host:port pairs.* **spring.redis.timeout**=0 *# Connection timeout in milliseconds.*

*# TRANSACTION (*[TransactionProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/transaction/TransactionProperties.java))

**spring.transaction.default-timeout**= *# Default transaction timeout in seconds.*

**spring.transaction.rollback-on-commit-failure**= *# Perform the rollback on commit failures.*

*# ---------------------------------------- # INTEGRATION PROPERTIES*

*# ----------------------------------------*

*# ACTIVEMQ (*[ActiveMQProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/activemq/ActiveMQProperties.java))

**spring.activemq.broker-url**= *# URL of the ActiveMQ broker. Auto-generated by default.*

**spring.activemq.close-timeout**=15000 *# Time to wait, in milliseconds, before considering a close complete.*

**spring.activemq.in-memory**=true *# Specify if the default broker URL should be in memory. Ignored if an explicit broker has been specified.*

**spring.activemq.non-blocking-redelivery**=false *# Do not stop message delivery before re-delivering messages from a rolled back transaction. This implies that message order will not be preserved when this is enabled.*

**spring.activemq.password**= *# Login password of the broker.*

**spring.activemq.send-timeout**=0 *# Time to wait, in milliseconds, on Message sends for a response. Set it to 0 to indicate to wait forever.*

**spring.activemq.user**= *# Login user of the broker.*

**spring.activemq.packages.trust-all**= *# Trust all packages.*

**spring.activemq.packages.trusted**= *# Comma-separated list of specific packages to trust (when not trusting all packages).*

**spring.activemq.pool.block-if-full**=true *# Block when a connection is requested and the pool is full. Set it to false to throw a "JMSException" instead.*

**spring.activemq.pool.block-if-full-timeout**=-1 *# Blocking period, in milliseconds, before throwing an exception if the pool is still full.*

**spring.activemq.pool.create-connection-on-startup**=true *# Create a connection on startup. Can be used to warm-up the pool on startup.*

**spring.activemq.pool.enabled**=false *# Whether a PooledConnectionFactory should be created instead of a regular ConnectionFactory.*

**spring.activemq.pool.expiry-timeout**=0 *# Connection expiration timeout in milliseconds.* **spring.activemq.pool.idle-timeout**=30000 *# Connection idle timeout in milliseconds.* **spring.activemq.pool.max-connections**=1 *# Maximum number of pooled connections.* **spring.activemq.pool.maximum-active-session-per-connection**=500 *# Maximum number of active sessions per*

*connection.*

**spring.activemq.pool.reconnect-on-exception**=true *# Reset the connection when a "JMXException" occurs.*

**spring.activemq.pool.time-between-expiration-check**=-1 *# Time to sleep, in milliseconds, between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.*

**spring.activemq.pool.use-anonymous-producers**=true *# Use only one anonymous "MessageProducer" instance. Set it to false to create one "MessageProducer" every time one is required.*

*# ARTEMIS (*[ArtemisProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/artemis/ArtemisProperties.java))

**spring.artemis.embedded.cluster-password**= *# Cluster password. Randomly generated on startup by default.*

**spring.artemis.embedded.data-directory**= *# Journal file directory. Not necessary if persistence is turned off.*

**spring.artemis.embedded.enabled**=true *# Enable embedded mode if the Artemis server APIs are available.*

**spring.artemis.embedded.persistent**=false *# Enable persistent store.* **spring.artemis.embedded.queues**= *# Comma-separated list of queues to create on startup.* **spring.artemis.embedded.server-id**= *# Server id. By default, an auto-incremented counter is used.* **spring.artemis.embedded.topics**= *# Comma-separated list of topics to create on startup.* **spring.artemis.host**=localhost *# Artemis broker host.*

**spring.artemis.mode**= *# Artemis deployment mode, auto-detected by default.* **spring.artemis.password**= *# Login password of the broker.* **spring.artemis.port**=61616 *# Artemis broker port.*

**spring.artemis.user**= *# Login user of the broker.*

*# SPRING BATCH (*[BatchProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/BatchProperties.java))

**spring.batch.initializer.enabled**= *# Create the required batch tables on startup if necessary. Enabled automatically if no custom table prefix is set or if a custom schema is configured.*

**spring.batch.job.enabled**=true *# Execute all Spring Batch jobs in the context on startup.*

**spring.batch.job.names**= *# Comma-separated list of job names to execute on startup (For instance*

*`job1,job2`). By default, all Jobs found in the context are executed.*

[**spring.batch.schema**=classpath:org/springframework/batch/core/schema-@@platform@@.sql](mailto:org/springframework/batch/core/schema-@@platform@@.sql) *# Path to the SQL file to use to initialize the database schema.*

**spring.batch.table-prefix**= *# Table prefix for all the batch meta-data tables.*

*# JMS (*[JmsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/JmsProperties.java))

**spring.jms.jndi-name**= *# Connection factory JNDI name. When set, takes precedence to others connection factory auto-configurations.*

**spring.jms.listener.acknowledge-mode**= *# Acknowledge mode of the container. By default, the listener is transacted with automatic acknowledgment.*

**spring.jms.listener.auto-startup**=true *# Start the container automatically on startup.* **spring.jms.listener.concurrency**= *# Minimum number of concurrent consumers.* **spring.jms.listener.max-concurrency**= *# Maximum number of concurrent consumers.* **spring.jms.pub-sub-domain**=false *# Specify if the default destination type is topic.*

**spring.jms.template.default-destination**= *# Default destination to use on send/receive operations that do not have a destination parameter.*

**spring.jms.template.delivery-delay**= *# Delivery delay to use for send calls in milliseconds.* **spring.jms.template.delivery-mode**= *# Delivery mode. Enable QoS when set.* **spring.jms.template.priority**= *# Priority of a message when sending. Enable QoS when set.* **spring.jms.template.qos-enabled**= *# Enable explicit QoS when sending a message.* **spring.jms.template.receive-timeout**= *# Timeout to use for receive calls in milliseconds.*

**spring.jms.template.time-to-live**= *# Time-to-live of a message when sending in milliseconds. Enable QoS when set.*

*# APACHE KAFKA (*[KafkaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/kafka/KafkaProperties.java))

**spring.kafka.bootstrap-servers**= *# Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.*

**spring.kafka.client-id**= *# Id to pass to the server when making requests; used for server-side logging.*

**spring.kafka.consumer.auto-commit-interval**= *# Frequency in milliseconds that the consumer offsets are auto-committed to Kafka if 'enable.auto.commit' true.*

**spring.kafka.consumer.auto-offset-reset**= *# What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server.*

**spring.kafka.consumer.bootstrap-servers**= *# Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.*

**spring.kafka.consumer.client-id**= *# Id to pass to the server when making requests; used for server-side logging.*

**spring.kafka.consumer.enable-auto-commit**= *# If true the consumer's offset will be periodically committed in the background.*

**spring.kafka.consumer.fetch-max-wait**= *# Maximum amount of time in milliseconds the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by "fetch.min.bytes".*

**spring.kafka.consumer.fetch-min-size**= *# Minimum amount of data the server should return for a fetch request in bytes.*

**spring.kafka.consumer.group-id**= *# Unique string that identifies the consumer group this consumer belongs to.*

**spring.kafka.consumer.heartbeat-interval**= *# Expected time in milliseconds between heartbeats to the consumer coordinator.*

**spring.kafka.consumer.key-deserializer**= *# Deserializer class for keys.*

**spring.kafka.consumer.max-poll-records**= *# Maximum number of records returned in a single call to poll().*

**spring.kafka.consumer.value-deserializer**= *# Deserializer class for values.*

**spring.kafka.listener.ack-count**= *# Number of records between offset commits when ackMode is "COUNT" or "COUNT\_TIME".*

**spring.kafka.listener.ack-mode**= *# Listener AckMode; see the spring-kafka documentation.*

**spring.kafka.listener.ack-time**= *# Time in milliseconds between offset commits when ackMode is "TIME" or "COUNT\_TIME".*

**spring.kafka.listener.concurrency**= *# Number of threads to run in the listener containers.* **spring.kafka.listener.poll-timeout**= *# Timeout in milliseconds to use when polling the consumer.* **spring.kafka.producer.acks**= *# Number of acknowledgments the producer requires the leader to have*

*received before considering a request complete.*

**spring.kafka.producer.batch-size**= *# Number of records to batch before sending.*

**spring.kafka.producer.bootstrap-servers**= *# Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.*

**spring.kafka.producer.buffer-memory**= *# Total bytes of memory the producer can use to buffer records waiting to be sent to the server.*

**spring.kafka.producer.client-id**= *# Id to pass to the server when making requests; used for server-side logging.*

**spring.kafka.producer.compression-type**= *# Compression type for all data generated by the producer.*

**spring.kafka.producer.key-serializer**= *# Serializer class for keys.* **spring.kafka.producer.retries**= *# When greater than zero, enables retrying of failed sends.* **spring.kafka.producer.value-serializer**= *# Serializer class for values.* **spring.kafka.properties.\***= *# Additional properties used to configure the client.* **spring.kafka.ssl.key-password**= *# Password of the private key in the key store file.* **spring.kafka.ssl.keystore-location**= *# Location of the key store file.* **spring.kafka.ssl.keystore-password**= *# Store password for the key store file.* **spring.kafka.ssl.truststore-location**= *# Location of the trust store file.* **spring.kafka.ssl.truststore-password**= *# Store password for the trust store file.* **spring.kafka.template.default-topic**= *# Default topic to which messages will be sent.*

*# RABBIT (*[RabbitProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/amqp/RabbitProperties.java))

**spring.rabbitmq.addresses**= *# Comma-separated list of addresses to which the client should connect.*

**spring.rabbitmq.cache.channel.checkout-timeout**= *# Number of milliseconds to wait to obtain a channel if the cache size has been reached.*

**spring.rabbitmq.cache.channel.size**= *# Number of channels to retain in the cache.* **spring.rabbitmq.cache.connection.mode**=channel *# Connection factory cache mode.* **spring.rabbitmq.cache.connection.size**= *# Number of connections to cache.* **spring.rabbitmq.connection-timeout**= *# Connection timeout, in milliseconds; zero for infinite.* **spring.rabbitmq.dynamic**=true *# Create an AmqpAdmin bean.*

**spring.rabbitmq.host**=localhost *# RabbitMQ host.*

**spring.rabbitmq.listener.simple.acknowledge-mode**= *# Acknowledge mode of container.* **spring.rabbitmq.listener.simple.auto-startup**=true *# Start the container automatically on startup.* **spring.rabbitmq.listener.simple.concurrency**= *# Minimum number of consumers.* **spring.rabbitmq.listener.simple.default-requeue-rejected**= *# Whether or not to requeue delivery failures;*

*default `true`.*

**spring.rabbitmq.listener.simple.idle-event-interval**= *# How often idle container events should be published in milliseconds.*

**spring.rabbitmq.listener.simple.max-concurrency**= *# Maximum number of consumers.*

**spring.rabbitmq.listener.simple.prefetch**= *# Number of messages to be handled in a single request. It should be greater than or equal to the transaction size (if used).*

**spring.rabbitmq.listener.simple.retry.enabled**=false *# Whether or not publishing retries are enabled.*

**spring.rabbitmq.listener.simple.retry.initial-interval**=1000 *# Interval between the first and second attempt to deliver a message.*

**spring.rabbitmq.listener.simple.retry.max-attempts**=3 *# Maximum number of attempts to deliver a message.* **spring.rabbitmq.listener.simple.retry.max-interval**=10000 *# Maximum interval between attempts.* **spring.rabbitmq.listener.simple.retry.multiplier**=1.0 *# A multiplier to apply to the previous delivery*

*retry interval.*

**spring.rabbitmq.listener.simple.retry.stateless**=true *# Whether or not retry is stateless or stateful.*

**spring.rabbitmq.listener.simple.transaction-size**= *# Number of messages to be processed in a transaction. For best results it should be less than or equal to the prefetch count.*

**spring.rabbitmq.password**= *# Login to authenticate against the broker.*

**spring.rabbitmq.port**=5672 *# RabbitMQ port.*

**spring.rabbitmq.publisher-confirms**=false *# Enable publisher confirms.*

**spring.rabbitmq.publisher-returns**=false *# Enable publisher returns.*

**spring.rabbitmq.requested-heartbeat**= *# Requested heartbeat timeout, in seconds; zero for none.*

**spring.rabbitmq.ssl.enabled**=false *# Enable SSL support.*

**spring.rabbitmq.ssl.key-store**= *# Path to the key store that holds the SSL certificate.* **spring.rabbitmq.ssl.key-store-password**= *# Password used to access the key store.* **spring.rabbitmq.ssl.trust-store**= *# Trust store that holds SSL certificates.* **spring.rabbitmq.ssl.trust-store-password**= *# Password used to access the trust store.* **spring.rabbitmq.ssl.algorithm**= *# SSL algorithm to use. By default configure by the rabbit client*

*library.*

**spring.rabbitmq.template.mandatory**=false *# Enable mandatory messages.* **spring.rabbitmq.template.receive-timeout**=0 *# Timeout for `receive()` methods.* **spring.rabbitmq.template.reply-timeout**=5000 *# Timeout for `sendAndReceive()` methods.* **spring.rabbitmq.template.retry.enabled**=false *# Set to true to enable retries in the `RabbitTemplate`.*

**spring.rabbitmq.template.retry.initial-interval**=1000 *# Interval between the first and second attempt to publish a message.*

**spring.rabbitmq.template.retry.max-attempts**=3 *# Maximum number of attempts to publish a message.* **spring.rabbitmq.template.retry.max-interval**=10000 *# Maximum number of attempts to publish a message.* **spring.rabbitmq.template.retry.multiplier**=1.0 *# A multiplier to apply to the previous publishing retry*

*interval.*

**spring.rabbitmq.username**= *# Login user to authenticate to the broker.*

**spring.rabbitmq.virtual-host**= *# Virtual host to use when connecting to the broker.*

*# ---------------------------------------- # ACTUATOR PROPERTIES*

*# ----------------------------------------*

*# ENDPOINTS (*[AbstractEndpoint](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/endpoint/AbstractEndpoint.java) subclasses) **endpoints.enabled**=true *# Enable endpoints.* **endpoints.sensitive**= *# Default endpoint sensitive setting.* **endpoints.actuator.enabled**=true *# Enable the endpoint.* **endpoints.actuator.path**= *# Endpoint URL path.*

**endpoints.actuator.sensitive**=false *# Enable security on the endpoint.* **endpoints.auditevents.enabled**= *# Enable the endpoint.* **endpoints.auditevents.path**= *# Endpoint path.* **endpoints.auditevents.sensitive**=false *# Enable security on the endpoint.* **endpoints.autoconfig.enabled**= *# Enable the endpoint.* **endpoints.autoconfig.id**= *# Endpoint identifier.* **endpoints.autoconfig.path**= *# Endpoint path.*

**endpoints.autoconfig.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.beans.enabled**= *# Enable the endpoint.* **endpoints.beans.id**= *# Endpoint identifier.* **endpoints.beans.path**= *# Endpoint path.*

**endpoints.beans.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.configprops.enabled**= *# Enable the endpoint.*

**endpoints.configprops.id**= *# Endpoint identifier.*

**endpoints.configprops.keys-to-sanitize**=password,secret,key,token,.\*credentials.\*,vcap\_services *# Keys that should be sanitized. Keys can be simple strings that the property ends with or regex expressions.*

**endpoints.configprops.path**= *# Endpoint path.*

**endpoints.configprops.sensitive**= *# Mark if the endpoint exposes sensitive information.* **endpoints.docs.curies.enabled**=false *# Enable the curie generation.* **endpoints.docs.enabled**=true *# Enable actuator docs endpoint.*

**endpoints.docs.path**=/docs *#* **endpoints.docs.sensitive**=false *#* **endpoints.dump.enabled**= *# Enable the endpoint.* **endpoints.dump.id**= *# Endpoint identifier.* **endpoints.dump.path**= *# Endpoint path.*

**endpoints.dump.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.env.enabled**= *# Enable the endpoint.*

**endpoints.env.id**= *# Endpoint identifier.*

**endpoints.env.keys-to-sanitize**=password,secret,key,token,.\*credentials.\*,vcap\_services *# Keys that should be sanitized. Keys can be simple strings that the property ends with or regex expressions.*

**endpoints.env.path**= *# Endpoint path.*

**endpoints.env.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.flyway.enabled**= *# Enable the endpoint.*

**endpoints.flyway.id**= *# Endpoint identifier.*

**endpoints.flyway.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.health.enabled**= *# Enable the endpoint.*

**endpoints.health.id**= *# Endpoint identifier.*

**endpoints.health.mapping.\***= *# Mapping of health statuses to HTTP status codes. By default, registered health statuses map to sensible defaults (i.e. UP maps to 200).*

**endpoints.health.path**= *# Endpoint path.*

**endpoints.health.sensitive**= *# Mark if the endpoint exposes sensitive information.* **endpoints.health.time-to-live**=1000 *# Time to live for cached result, in milliseconds.* **endpoints.heapdump.enabled**= *# Enable the endpoint.*

**endpoints.heapdump.path**= *# Endpoint path.*

**endpoints.heapdump.sensitive**= *# Mark if the endpoint exposes sensitive information.* **endpoints.hypermedia.enabled**=false *# Enable hypermedia support for endpoints.* **endpoints.info.enabled**= *# Enable the endpoint.*

**endpoints.info.id**= *# Endpoint identifier.*

**endpoints.info.path**= *# Endpoint path.*

**endpoints.info.sensitive**= *# Mark if the endpoint exposes sensitive information.* **endpoints.jolokia.enabled**=true *# Enable Jolokia endpoint.* **endpoints.jolokia.path**=/jolokia *# Endpoint URL path.* **endpoints.jolokia.sensitive**=true *# Enable security on the endpoint.* **endpoints.liquibase.enabled**= *# Enable the endpoint.*

**endpoints.liquibase.id**= *# Endpoint identifier.*

**endpoints.liquibase.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.logfile.enabled**=true *# Enable the endpoint.* **endpoints.logfile.external-file**= *# External Logfile to be accessed.* **endpoints.logfile.path**=/logfile *# Endpoint URL path.* **endpoints.logfile.sensitive**=true *# Enable security on the endpoint.* **endpoints.loggers.enabled**=true *# Enable the endpoint.* **endpoints.loggers.id**= *# Endpoint identifier.* **endpoints.loggers.path**=/logfile *# Endpoint path.*

**endpoints.loggers.sensitive**=true *# Mark if the endpoint exposes sensitive information.*

**endpoints.mappings.enabled**= *# Enable the endpoint.* **endpoints.mappings.id**= *# Endpoint identifier.* **endpoints.mappings.path**= *# Endpoint path.*

**endpoints.mappings.sensitive**= *# Mark if the endpoint exposes sensitive information.* **endpoints.metrics.enabled**= *# Enable the endpoint.* **endpoints.metrics.filter.enabled**=true *# Enable the metrics servlet filter.*

**endpoints.metrics.filter.gauge-submissions**=merged *# Http filter gauge submissions (merged, per-http- method)*

**endpoints.metrics.filter.counter-submissions**=merged *# Http filter counter submissions (merged, per-http- method)*

**endpoints.metrics.id**= *# Endpoint identifier.*

**endpoints.metrics.path**= *# Endpoint path.*

**endpoints.metrics.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.shutdown.enabled**= *# Enable the endpoint.*

**endpoints.shutdown.id**= *# Endpoint identifier.*

**endpoints.shutdown.path**= *# Endpoint path.*

**endpoints.shutdown.sensitive**= *# Mark if the endpoint exposes sensitive information.*

**endpoints.trace.enabled**= *# Enable the endpoint.* **endpoints.trace.filter.enabled**=true *# Enable the trace servlet filter.* **endpoints.trace.id**= *# Endpoint identifier.*

**endpoints.trace.path**= *# Endpoint path.*

**endpoints.trace.sensitive**= *# Mark if the endpoint exposes sensitive information.*

*# ENDPOINTS CORS CONFIGURATION (*[EndpointCorsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointCorsProperties.java))

**endpoints.cors.allow-credentials**= *# Set whether credentials are supported. When not set, credentials are not supported.*

**endpoints.cors.allowed-headers**= *# Comma-separated list of headers to allow in a request. '\*' allows all headers.*

**endpoints.cors.allowed-methods**=GET *# Comma-separated list of methods to allow. '\*' allows all methods.*

**endpoints.cors.allowed-origins**= *# Comma-separated list of origins to allow. '\*' allows all origins. When not set, CORS support is disabled.*

**endpoints.cors.exposed-headers**= *# Comma-separated list of headers to include in a response.*

**endpoints.cors.max-age**=1800 *# How long, in seconds, the response from a pre-flight request can be cached by clients.*

*# JMX ENDPOINT (*[EndpointMBeanExportProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointMBeanExportProperties.java))

**endpoints.jmx.domain**= *# JMX domain name. Initialized with the value of 'spring.jmx.default-domain' if set.*

**endpoints.jmx.enabled**=true *# Enable JMX export of all endpoints.*

**endpoints.jmx.static-names**= *# Additional static properties to append to all ObjectNames of MBeans representing Endpoints.*

**endpoints.jmx.unique-names**=false *# Ensure that ObjectNames are modified in case of conflict.*

*# JOLOKIA (*[JolokiaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/JolokiaProperties.java))

**jolokia.config.\***= *# See Jolokia manual*

*# MANAGEMENT HTTP SERVER (*[ManagementServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ManagementServerProperties.java))

**management.add-application-context-header**=true *# Add the "X-Application-Context" HTTP header in each response.*

**management.address**= *# Network address that the management endpoints should bind to.* **management.context-path**= *# Management endpoint context-path. For instance `/actuator`* **management.cloudfoundry.enabled**= *# Enable extended Cloud Foundry actuator endpoints*

**management.cloudfoundry.skip-ssl-validation**= *# Skip SSL verification for Cloud Foundry actuator endpoint security calls*

**management.port**= *# Management endpoint HTTP port. Uses the same port as the application by default. Configure a different port to use management-specific SSL.*

**management.security.enabled**=true *# Enable security.*

**management.security.roles**=ACTUATOR *# Comma-separated list of roles that can access the management endpoint.*

**management.security.sessions**=stateless *# Session creating policy to use (always, never, if\_required, stateless).*

**management.ssl.ciphers**= *# Supported SSL ciphers. Requires a custom management.port.*

**management.ssl.client-auth**= *# Whether client authentication is wanted ("want") or needed ("need"). Requires a trust store. Requires a custom management.port.*

**management.ssl.enabled**= *# Enable SSL support. Requires a custom management.port.* **management.ssl.enabled-protocols**= *# Enabled SSL protocols. Requires a custom management.port.* **management.ssl.key-alias**= *# Alias that identifies the key in the key store. Requires a custom*

*management.port.*

**management.ssl.key-password**= *# Password used to access the key in the key store. Requires a custom management.port.*

**management.ssl.key-store**= *# Path to the key store that holds the SSL certificate (typically a jks file). Requires a custom management.port.*

**management.ssl.key-store-password**= *# Password used to access the key store. Requires a custom management.port.*

**management.ssl.key-store-provider**= *# Provider for the key store. Requires a custom management.port.* **management.ssl.key-store-type**= *# Type of the key store. Requires a custom management.port.* **management.ssl.protocol**=TLS *# SSL protocol to use. Requires a custom management.port.* **management.ssl.trust-store**= *# Trust store that holds SSL certificates. Requires a custom*

*management.port.*

**management.ssl.trust-store-password**= *# Password used to access the trust store. Requires a custom management.port.*

**management.ssl.trust-store-provider**= *# Provider for the trust store. Requires a custom management.port.*

**management.ssl.trust-store-type**= *# Type of the trust store. Requires a custom management.port. # HEALTH INDICATORS*

**management.health.db.enabled**=true *# Enable database health check.* **management.health.cassandra.enabled**=true *# Enable cassandra health check.* **management.health.couchbase.enabled**=true *# Enable couchbase health check.* **management.health.defaults.enabled**=true *# Enable default health indicators.* **management.health.diskspace.enabled**=true *# Enable disk space health check.* **management.health.diskspace.path**= *# Path used to compute the available disk space.* **management.health.diskspace.threshold**=0 *# Minimum disk space that should be available, in bytes.* **management.health.elasticsearch.enabled**=true *# Enable elasticsearch health check.* **management.health.elasticsearch.indices**= *# Comma-separated index names.* **management.health.elasticsearch.response-timeout**=100 *# The time, in milliseconds, to wait for a response*

*from the cluster.*

**management.health.jms.enabled**=true *# Enable JMS health check.* **management.health.ldap.enabled**=true *# Enable LDAP health check.* **management.health.mail.enabled**=true *# Enable Mail health check.* **management.health.mongo.enabled**=true *# Enable MongoDB health check.* **management.health.rabbit.enabled**=true *# Enable RabbitMQ health check.* **management.health.redis.enabled**=true *# Enable Redis health check.* **management.health.solr.enabled**=true *# Enable Solr health check.*

**management.health.status.order**=DOWN, OUT\_OF\_SERVICE, UP, UNKNOWN *# Comma-separated list of health statuses in order of severity.*

*# INFO CONTRIBUTORS (*[InfoContributorProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/InfoContributorProperties.java)) **management.info.build.enabled**=true *# Enable build info.* **management.info.defaults.enabled**=true *# Enable default info contributors.* **management.info.env.enabled**=true *# Enable environment info.* **management.info.git.enabled**=true *# Enable git info.* **management.info.git.mode**=simple *# Mode to use to expose git information.*

*# REMOTE SHELL (*[ShellProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ShellProperties.java))

**management.shell.auth.type**=simple *# Authentication type. Auto-detected according to the environment.*

**management.shell.auth.jaas.domain**=my-domain *# JAAS domain.*

**management.shell.auth.key.path**= *# Path to the authentication key. This should point to a valid ".pem" file.*

**management.shell.auth.simple.user.name**=user *# Login user.* **management.shell.auth.simple.user.password**= *# Login password.* **management.shell.auth.spring.roles**=ACTUATOR *# Comma-separated list of required roles to login to the*

*CRaSH console.*

**management.shell.command-path-patterns**=classpath\*:/commands/\*\*,classpath\*:/crash/commands/\*\* *# Patterns to use to look for commands.*

**management.shell.command-refresh-interval**=-1 *# Scan for changes and update the command if necessary (in seconds).*

**management.shell.config-path-patterns**=classpath\*:/crash/\* *# Patterns to use to look for configurations.* **management.shell.disabled-commands**=jpa\*,jdbc\*,jndi\* *# Comma-separated list of commands to disable.* **management.shell.disabled-plugins**= *# Comma-separated list of plugins to disable. Certain plugins are*

*disabled by default based on the environment.*

**management.shell.ssh.auth-timeout** = *# Number of milliseconds after user will be prompted to login again.*

**management.shell.ssh.enabled**=true *# Enable CRaSH SSH support.*

**management.shell.ssh.idle-timeout** = *# Number of milliseconds after which unused connections are closed.*

**management.shell.ssh.key-path**= *# Path to the SSH server key.*

**management.shell.ssh.port**=2000 *# SSH port.*

**management.shell.telnet.enabled**=false *# Enable CRaSH telnet support. Enabled by default if the TelnetPlugin is available.*

**management.shell.telnet.port**=5000 *# Telnet port.*

*# TRACING (*[TraceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/trace/TraceProperties.java))

**management.trace.include**=request-headers,response-headers,cookies,errors *# Items to be included in the trace.*

*# METRICS EXPORT (*[MetricExportProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/export/MetricExportProperties.java))

**spring.metrics.export.aggregate.key-pattern**= *# Pattern that tells the aggregator what to do with the keys from the source repository.*

**spring.metrics.export.aggregate.prefix**= *# Prefix for global repository if active.*

**spring.metrics.export.delay-millis**=5000 *# Delay in milliseconds between export ticks. Metrics are exported to external sources on a schedule with this delay.*

**spring.metrics.export.enabled**=true *# Flag to enable metric export (assuming a MetricWriter is available).*

**spring.metrics.export.excludes**= *# List of patterns for metric names to exclude. Applied after the includes.*

**spring.metrics.export.includes**= *# List of patterns for metric names to include.* **spring.metrics.export.redis.key**=keys.spring.metrics *# Key for redis repository export (if active).* **spring.metrics.export.redis.prefix**=spring.metrics *# Prefix for redis repository if active.*

**spring.metrics.export.send-latest**= *# Flag to switch off any available optimizations based on not exporting unchanged metric values.*

**spring.metrics.export.statsd.host**= *# Host of a statsd server to receive exported metrics.* **spring.metrics.export.statsd.port**=8125 *# Port of a statsd server to receive exported metrics.* **spring.metrics.export.statsd.prefix**= *# Prefix for statsd exported metrics.* **spring.metrics.export.triggers.\***= *# Specific trigger properties per MetricWriter bean name.*

*# ---------------------------------------- # DEVTOOLS PROPERTIES*

*# ----------------------------------------*

*# DEVTOOLS (*[DevToolsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-devtools/src/main/java/org/springframework/boot/devtools/autoconfigure/DevToolsProperties.java))

**spring.devtools.livereload.enabled**=true *# Enable a livereload.com compatible server.*

**spring.devtools.livereload.port**=35729 *# Server port.*

**spring.devtools.restart.additional-exclude**= *# Additional patterns that should be excluded from triggering a full restart.*

**spring.devtools.restart.additional-paths**= *# Additional paths to watch for changes.*

**spring.devtools.restart.enabled**=true *# Enable automatic restart.*

**spring.devtools.restart.exclude**=META-INF/maven/\*\*,META-INF/resources/\*\*,resources/\*\*,static/\*\*,public/

\*\*,templates/\*\*,\*\*/\*Test.class,\*\*/\*Tests.class,git.properties *# Patterns that should be excluded from triggering a full restart.*

**spring.devtools.restart.poll-interval**=1000 *# Amount of time (in milliseconds) to wait between polling for classpath changes.*

**spring.devtools.restart.quiet-period**=400 *# Amount of quiet time (in milliseconds) required without any classpath changes before a restart is triggered.*

**spring.devtools.restart.trigger-file**= *# Name of a specific file that when changed will trigger the restart check. If not specified any classpath file change will trigger the restart.*

*# REMOTE DEVTOOLS (*[RemoteDevToolsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-devtools/src/main/java/org/springframework/boot/devtools/autoconfigure/RemoteDevToolsProperties.java))

**spring.devtools.remote.context-path**=/.~~spring-boot!~ *# Context path used to handle the remote connection.*

**spring.devtools.remote.debug.enabled**=true *# Enable remote debug support.* **spring.devtools.remote.debug.local-port**=8000 *# Local remote debug server port.* **spring.devtools.remote.proxy.host**= *# The host of the proxy to use to connect to the remote application.* **spring.devtools.remote.proxy.port**= *# The port of the proxy to use to connect to the remote application.* **spring.devtools.remote.restart.enabled**=true *# Enable remote restart.*

**spring.devtools.remote.secret**= *# A shared secret required to establish a connection (required to enable remote support).*

**spring.devtools.remote.secret-header-name**=X-AUTH-TOKEN *# HTTP header used to transfer the shared secret.*

*# ---------------------------------------- # TESTING PROPERTIES*

*# ----------------------------------------*

**spring.test.database.replace**=any *# Type of existing DataSource to replace.*

**spring.test.mockmvc.print**=default *# MVC Print option.*

**Appendix B. Configuration meta-data**

Spring Boot jars are shipped with meta-data files that provide details of all supported configuration properties. The files are designed to allow IDE developers to offer contextual help and “code completion” as users are working with application.properties or application.yml files.

The majority of the meta-data file is generated automatically at compile time by processing all items annotated with @ConfigurationProperties. However, it is possible to [write part of the meta-data](#_bookmark588) [manually](#_bookmark588) for corner cases or more advanced use cases.

## Meta-data format

Configuration meta-data files are located inside jars under META-INF/spring-configuration- metadata.json They use a simple JSON format with items categorized under either “groups” or “properties” and additional values hint categorized under "hints":

{***"groups"***: **[**

**{**

***"name"***: ***"server"*,**

***"type"***: ***"org.springframework.boot.autoconfigure.web.ServerProperties"*, *"sourceType"***: ***"org.springframework.boot.autoconfigure.web.ServerProperties"***

**},**

**{**

***"name"***: ***"spring.jpa.hibernate"*,**

***"type"***: ***"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"*, *"sourceType"***: ***"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties"*, *"sourceMethod"***: ***"getHibernate()"***

**}**

...

],***"properties"***: **[**

**{**

***"name"***: ***"server.port"*, *"type"***: ***"java.lang.Integer"*,**

***"sourceType"***: ***"org.springframework.boot.autoconfigure.web.ServerProperties"***

**},**

**{**

***"name"***: ***"server.servlet-path"*, *"type"***: ***"java.lang.String"*,**

***"sourceType"***: ***"org.springframework.boot.autoconfigure.web.ServerProperties"*, *"defaultValue"***: ***"/"***

**},**

**{**

***"name"***: ***"spring.jpa.hibernate.ddl-auto"*, *"type"***: ***"java.lang.String"*,**

***"description"***: ***"DDL mode. This is actually a shortcut for the \"hibernate.hbm2ddl.auto\" property."*,**

***"sourceType"***: ***"org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"***

**}**

...

],***"hints"***: **[**

**{**

***"name"***: ***"spring.jpa.hibernate.ddl-auto"*, *"values"***: **[**

**{**

***"value"***: ***"none"*,**

***"description"***: ***"Disable DDL handling."***

**},**

**{**

***"value"***: ***"validate"*,**

***"description"***: ***"Validate the schema, make no changes to the database."***

**},**

**{**

***"value"***: ***"update"*,**

***"description"***: ***"Update the schema if necessary."***

**},**

**{**

***"value"***: ***"create"*,**

***"description"***: ***"Create the schema and destroy previous data."***

**},**

**{**

***"value"***: ***"create-drop"*,**

***"description"***: ***"Create and then destroy the schema at the end of the session."***

**}**

**]**

**}**

]**}**

Each “property” is a configuration item that the user specifies with a given value. For example server.port and server.servlet-path might be specified in application.properties as follows:

**server.port**=9090 **server.servlet-path**=/home

The “groups” are higher level items that don’t themselves specify a value, but instead provide a contextual grouping for properties. For example the server.port and server.servlet-path properties are part of the server group.

**Note**

It is not required that every “property” has a “group”, some properties might just exist in their own right.

Finally, “hints” are additional information used to assist the user in configuring a given property. When configuring the spring.jpa.hibernate.ddl-auto property, a tool can use it to offer some auto- completion help for the none, validate, update, create and create-drop values.

### Group Attributes

The JSON object contained in the groups array can contain the following attributes:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Purpose** |
| name type | String String | The full name of the group. This attribute is mandatory.  The class name of the data type of the group. For example, if the group was based on a class annotated with  @ConfigurationProperties the attribute would contain the fully qualified name of that class. If it was based on a @Bean method, it would be the return type of that method. The attribute may be omitted if the type is not known. |
| description | String | A short description of the group that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |
| sourceType | String | The class name of the source that contributed this group. For example, if the group was based on a @Bean method annotated |

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Purpose** |
| sourceMetho | d String | with @ConfigurationProperties this attribute would contain the fully qualified name of the @Configuration class containing the method. The attribute may be omitted if the source type is not known.  The full name of the method (include parenthesis and argument types) that contributed this group. For example, the name of a @ConfigurationProperties annotated @Bean method. May be omitted if the source method is not known. |

### Property Attributes

The JSON object contained in the properties array can contain the following attributes:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Purpose** |
| name | String | The full name of the property. Names are in lowercase dashed form (e.g. server.servlet-path). This attribute is mandatory. |
| type | String | The full signature of the data type of the property. For example, java.lang.String but also a full generic type such as java.util.Map<java.util.String,acme.MyEnum>. This attribute can be used to guide the user as to the types of values that they can enter. For consistency, the type of a primitive is specified using its wrapper counterpart, i.e. boolean becomes java.lang.Boolean. Note that this class may be a complex type that gets converted from a String as values are bound. May be omitted if the type is not known. |
| description  sourceType | String  String | A short description of the group that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.).  The class name of the source that contributed this property.  For example, if the property was from a class annotated with @ConfigurationProperties this attribute would contain the fully qualified name of that class. May be omitted if the source type is not known. |
| defaultValu | e Object | The default value which will be used if the property is not specified. Can also be an array of value(s) if the type of the property is an array. May be omitted if the default value is not known. |
| deprecation | Deprecation | Specify if the property is deprecated. May be omitted if the field is not deprecated or if that information is not known. See below for more details. |

The JSON object contained in the deprecation attribute of each properties element can contain the following attributes:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Purpose** |
| level  reason | String  String | The level of deprecation, can be either warning (default) or error. When a property has a warning deprecation level it should still be bound in the environment. When it has an error deprecation level however, the property is no longer managed and will not be bound.  A short description of the reason why the property was deprecated. May be omitted if no reason is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |
| replacement | String | The full name of the property that is *replacing* this deprecated property. May be omitted if there is no replacement for this property. |

**Note**

Prior to Spring Boot 1.3, a single deprecated boolean attribute can be used instead of the deprecation element. This is still supported in a deprecated fashion and should no longer be used. If no reason and replacement are available, an empty deprecation object should be set.

Deprecation can also be specified declaratively in code by adding the @DeprecatedConfigurationProperty annotation to the getter exposing the deprecated property. For instance, let’s assume the app.foo.target property was confusing and was renamed to app.foo.name

@ConfigurationProperties("app.foo")

**public class** FooProperties {

**private** String name;

**public** String getName() { ... }

**public void** setName(String name) { ... }

@DeprecatedConfigurationProperty(replacement = "app.foo.name") @Deprecated

**public** String getTarget() {

**return** getName();

}

@Deprecated

**public void** setTarget(String target) { setName(target);

}

}

**Note**

There is no way to set a level as warning is always assumed since code is still handling the property.

The code above makes sure that the deprecated property still works (delegating to the name property behind the scenes). Once the getTarget and setTarget methods can be removed from your public

API, the automatic deprecation hint in the meta-data will go away as well. If you want to keep a hint, adding manual meta-data with an error deprecation level ensures that users are still informed about that property and is particularly useful when a replacement is provided.

### Hint Attributes

The JSON object contained in the hints array can contain the following attributes:

|  |  |  |
| --- | --- | --- |
| **Name**  name | **Type**  String | **Purpose**  The full name of the property that this hint refers to. Names are in lowercase dashed form (e.g. server.servlet-path). If the property refers to a map (e.g. system.contexts) the hint either applies to the *keys* of the map (system.context.keys) or the  values (system.context.values). This attribute is mandatory. |
| values  providers | ValueHint[]  ValueProvider[] | A list of valid values as defined by the ValueHint object (see below). Each entry defines the value and may have a description  A list of providers as defined by the ValueProvider object (see below). Each entry defines the name of the provider and its parameters, if any. |

The JSON object contained in the values attribute of each hint element can contain the following attributes:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Purpose** |
| value | Object | A valid value for the element to which the hint refers to. Can also be an array of value(s) if the type of the property is an array. This attribute is mandatory. |
| description | String | A short description of the value that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |

The JSON object contained in the providers attribute of each hint element can contain the following attributes:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Purpose** |
| name  parameters | String  JSON object | The name of the provider to use to offer additional content assistance for the element to which the hint refers to.  Any additional parameter that the provider supports (check the documentation of the provider for more details). |

### Repeated meta-data items

It is perfectly acceptable for “property” and “group” objects with the same name to appear multiple times within a meta-data file. For example, you could bind two separate classes to the same prefix, with each

potentially offering overlap of property names. While this is not supposed to be a frequent scenario, consumers of meta-data should take care to ensure that they support such scenarios.

## Providing manual hints

To improve the user experience and further assist the user in configuring a given property, you can provide additional meta-data that:

1. Describes the list of potential values for a property.
2. Associates a provider to attach a well-defined semantic to a property so that a tool can discover the list of potential values based on the project’s context.

### Value hint

The name attribute of each hint refers to the name of a property. In the initial example above, we provide 5 values for the spring.jpa.hibernate.ddl-auto property: none, validate, update, create and create-drop. Each value may have a description as well.

If your property is of type Map, you can provide hints for both the keys and the values (but not for the map itself). The special .keys and .values suffixes must be used to refer to the keys and the values respectively.

Let’s assume a foo.contexts that maps magic String values to an integer:

@ConfigurationProperties("foo")

**public class** FooProperties {

**private** Map<String,Integer> contexts;

*// getters and setters*

}

The magic values are foo and bar for instance. In order to offer additional content assistance for the keys, you could add the following to [the manual meta-data of the module](#_bookmark588):

{***"hints"***: **[**

**{**

***"name"***: ***"foo.contexts.keys"*, *"values"***: **[**

**{**

***"value"***: ***"foo"***

**},**

**{**

***"value"***: ***"bar"***

**}**

**]**

**}**

]**}**

**Note**

Of course, you should have an Enum for those two values instead. This is by far the most effective approach to auto-completion if your IDE supports it.

### Value provider

Providers are a powerful way of attaching semantics to a property. We define in the section below the official providers that you can use for your own hints. Bare in mind however that your favorite IDE may implement some of these or none of them. It could eventually provide its own as well.

**Note**

As this is a new feature, IDE vendors will have to catch up with this new feature. The table below summarizes the list of supported providers:

|  |  |
| --- | --- |
| **Name**  any | **Description**  Permit any additional value to be provided. |
| class-reference  handle-as | Auto-complete the classes available in the project. Usually constrained by a base class that is specified via the target parameter.  Handle the property as if it was defined by the type defined via the mandatory target parameter. |
| logger-name | Auto-complete valid logger names. Typically, package and class names available in the current project can be auto-completed. |
| spring-bean-reference | Auto-complete the available bean names in the current project. Usually constrained by a base class that is specified via the target parameter. |
| spring-profile-name | Auto-complete the available Spring profile names in the project. |

**Tip**

No more than one provider can be active for a given property but you can specify several providers if they can all manage the property *in some ways*. Make sure to place the most powerful provider first as the IDE must use the first one in the JSON section it can handle. If no provider for a given property is supported, no special content assistance is provided either.

#### Any

The **any** provider permits any additional values to be provided. Regular value validation based on the property type should be applied if this is supported.

This provider will be typically used if you have a list of values and any extra values are still to be considered as valid.

The example below offers on and off as auto-completion values for system.state; any other value is also allowed:

{***"hints"***: **[**

**{**

***"name"***: ***"system.state"*, *"values"***: **[**

**{**

***"value"***: ***"on"***

**},**

**{**

***"value"***: ***"off"***

**}**

]**,**

***"providers"***: **[**

**{**

***"name"***: ***"any"***

**}**

**]**

**}**

]**}**

#### Class reference

The **class-reference** provider auto-completes classes available in the project. This provider supports these parameters:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Default value Description** |
| target  concrete | String  (Class)  boolean | *none* The fully qualified name of the class that should be assignable to the chosen value. Typically used to filter out non candidate classes. Note that this information can be provided by the type itself by exposing a class with the appropriate upper bound.  true Specify if only concrete classes are to be considered as valid candidates. |

The meta-data snippet below corresponds to the standard server.jsp-servlet.class-name

property that defines the JspServlet class name to use:

{***"hints"***: **[**

**{**

***"name"***: ***"server.jsp-servlet.class-name"*, *"providers"***: **[**

**{**

***"name"***: ***"class-reference"*, *"parameters"***: **{**

***"target"***: ***"javax.servlet.http.HttpServlet"***

**}**

**}**

**]**

**}**

]**}**

#### Handle As

The **handle-as** provider allows you to substitute the type of the property to a more high-level type. This typically happens when the property has a java.lang.String type because you don’t want your configuration classes to rely on classes that may not be on the classpath. This provider supports these parameters:

|  |  |  |
| --- | --- | --- |
| **Parameter**  **target** | **Type**  String  (Class) | **Default value Description**  *none* The fully qualified name of the type to consider for the property. This parameter is mandatory. |

The following types can be used:

* Any java.lang.Enum that lists the possible values for the property (By all means, try to define the property with the Enum type instead as no further hint should be required for the IDE to auto-complete the values).
* java.nio.charset.Charset: auto-completion of charset/encoding values (e.g. UTF-8)
* java.util.Locale: auto-completion of locales (e.g. en\_US)
* org.springframework.util.MimeType: auto-completion of content type values (e.g. text/ plain)
* org.springframework.core.io.Resource: auto-completion of Spring’s Resource abstraction to refer to a file on the filesystem or on the classpath. (e.g. classpath:/foo.properties)

**Note**

If multiple values can be provided, use a Collection or *Array* type to teach the IDE about it.

The meta-data snippet below corresponds to the standard liquibase.change-log property that defines the path to the changelog to use. It is actually used internally as a org.springframework.core.io.Resource but cannot be exposed as such as we need to keep the original String value to pass it to the Liquibase API.

{***"hints"***: **[**

**{**

***"name"***: ***"liquibase.change-log"*, *"providers"***: **[**

**{**

***"name"***: ***"handle-as"*, *"parameters"***: **{**

***"target"***: ***"org.springframework.core.io.Resource"***

**}**

**}**

**]**

**}**

]**}**

#### Logger name

The **logger-name** provider auto-completes valid logger names. Typically, package and class names available in the current project can be auto-completed. Specific frameworks may have extra magic logger names that could be supported as well.

Since a logger name can be any arbitrary name, really, this provider should allow any value but could highlight valid packages and class names that are not available in the project’s classpath.

The meta-data snippet below corresponds to the standard logging.level property, keys are *logger names* and values correspond to the standard log levels or any custom level:

{***"hints"***: **[**

**{**

***"name"***: ***"logging.level.keys"*, *"values"***: **[**

**{**

***"value"***: ***"root"*,**

***"description"***: ***"Root logger used to assign the default logging level."***

**}**

]**,**

***"providers"***: **[**

**{**

***"name"***: ***"logger-name"***

**}**

**]**

**},**

**{**

***"name"***: ***"logging.level.values"*, *"values"***: **[**

**{**

***"value"***: ***"trace"***

**},**

**{**

***"value"***: ***"debug"***

**},**

**{**

***"value"***: ***"info"***

**},**

**{**

***"value"***: ***"warn"***

**},**

**{**

***"value"***: ***"error"***

**},**

**{**

***"value"***: ***"fatal"***

**},**

**{**

***"value"***: ***"off"***

**}**

]**,**

***"providers"***: **[**

**{**

***"name"***: ***"any"***

**}**

**]**

**}**

]**}**

#### Spring bean reference

The **spring-bean-reference** provider auto-completes the beans that are defined in the configuration of the current project. This provider supports these parameters:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Default value Description** |
| target | String  (Class) | *none* The fully qualified name of the bean class that should be assignable to the candidate. Typically used to filter out non candidate beans. |

The meta-data snippet below corresponds to the standard spring.jmx.server property that defines the name of the MBeanServer bean to use:

{***"hints"***: **[**

**{**

***"name"***: ***"spring.jmx.server"*, *"providers"***: **[**

**{**

***"name"***: ***"spring-bean-reference"*, *"parameters"***: **{**

***"target"***: ***"javax.management.MBeanServer"***

**}**

**}**

**]**

**}**

]**}**

**Note**

The binder is not aware of the meta-data so if you provide that hint, you will still need to transform the bean name into an actual Bean reference using the ApplicationContext.

#### Spring profile name

The **spring-profile-name** provider auto-completes the Spring profiles that are defined in the configuration of the current project.

The meta-data snippet below corresponds to the standard spring.profiles.active property that defines the name of the Spring profile(s) to enable:

{***"hints"***: **[**

**{**

***"name"***: ***"spring.profiles.active"*, *"providers"***: **[**

**{**

***"name"***: ***"spring-profile-name"***

**}**

**]**

**}**

]**}**

## Generating your own meta-data using the annotation processor

You can easily generate your own configuration meta-data file from items annotated with @ConfigurationProperties by using the spring-boot-configuration-processor jar. The jar includes a Java annotation processor which is invoked as your project is compiled. To use the processor, simply include spring-boot-configuration-processor as an optional dependency, for example with Maven you would add:

**<dependency>**

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

**<artifactId>**spring-boot-configuration-processor**</artifactId>**

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

**</dependency>**

With Gradle, you can use the [propdeps-plugin](https://github.com/spring-gradle-plugins/propdeps-plugin) and specify:

dependencies {

optional ***"org.springframework.boot:spring-boot-configuration-processor"***

}

compileJava.dependsOn(processResources)

**Note**

You need to add compileJava.dependsOn(processResources) to your build to ensure that resources are processed before code is compiled. Without this directive any additional- spring-configuration-metadata.json files will not be processed.

The processor will pick up both classes and methods that are annotated with @ConfigurationProperties. The Javadoc for field values within configuration classes will be used to populate the description attribute.

**Note**

You should only use simple text with @ConfigurationProperties field Javadoc since they are not processed before being added to the JSON.

Properties are discovered via the presence of standard getters and setters with special handling for collection types (that will be detected even if only a getter is present). The annotation processor also supports the use of the @Data, @Getter and @Setter lombok annotations.

**Note**

If you are using AspectJ in your project, you need to make sure that the annotation processor only runs once. There are several ways to do this: with Maven, you can configure the maven-apt- plugin explicitly and add the dependency to the annotation processor only there. You could also let the AspectJ plugin run all the processing and disable annotation processing in the maven- compiler-plugin configuration:

**<plugin>**

**<groupId>**org.apache.maven.plugins**</groupId>**

**<artifactId>**maven-compiler-plugin**</artifactId>**

**<configuration>**

**<proc>**none**</proc>**

**</configuration>**

**</plugin>**

### Nested properties

The annotation processor will automatically consider inner classes as nested properties. For example, the following class:

@ConfigurationProperties(prefix="server")

**public class** ServerProperties { **private** String name; **private** Host host;

*// ... getter and setters*

**private static class** Host { **private** String ip; **private int** port;

*// ... getter and setters*

}

}

Will produce meta-data information for server.name, server.host.ip and server.host.port properties. You can use the @NestedConfigurationProperty annotation on a field to indicate that a regular (non-inner) class should be treated as if it were nested.

**Tip**

This has no effect on collections and maps as those types are automatically identified and a single meta-data property is generated for each of them.

### Adding additional meta-data

Spring Boot’s configuration file handling is quite flexible; and it is often the case that properties may exist that are not bound to a @ConfigurationProperties bean. You may also need to tune

some attributes of an existing key. To support such cases and allow you to provide custom "hints", the annotation processor will automatically merge items from META-INF/additional-spring- configuration-metadata.json into the main meta-data file.

If you refer to a property that has been detected automatically, the description, default value and deprecation information are overridden if specified. If the manual property declaration is not identified in the current module, it is added as a brand new property.

The format of the additional-spring-configuration-metadata.json file is exactly the same as the regular spring-configuration-metadata.json. The additional properties file is optional, if you don’t have any additional properties, simply don’t add it.

**Appendix C. Auto-configuration classes**

Here is a list of all auto-configuration classes provided by Spring Boot with links to documentation and source code. Remember to also look at the autoconfig report in your application for more details of which features are switched on. (start the app with --debug or -Ddebug, or in an Actuator application use the autoconfig endpoint).

## From the “spring-boot-autoconfigure” module

The following auto-configuration classes are from the spring-boot-autoconfigure module:

|  |  |
| --- | --- |
| **Configuration Class** | **Links** |
| [ActiveMQAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/activemq/ActiveMQAutoConfiguration.java)  [AopAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/aop/AopAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jms/activemq/ActiveMQAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/aop/AopAutoConfiguration.html) |
| [ArtemisAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/artemis/ArtemisAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jms/artemis/ArtemisAutoConfiguration.html) |
| [BatchAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/BatchAutoConfiguration.java)  [CacheAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cache/CacheAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/batch/BatchAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/cache/CacheAutoConfiguration.html) |
| [CassandraAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cassandra/CassandraAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/cassandra/CassandraAutoConfiguration.html) |
| [CassandraDataAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/cassandra/CassandraDataAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/cassandra/CassandraDataAutoConfiguration.html) |
| [CassandraRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/cassandra/CassandraRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/cassandra/CassandraRepositoriesAutoConfiguration.html) |
| [CloudAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cloud/CloudAutoConfiguration.java)  [ConfigurationPropertiesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/context/ConfigurationPropertiesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/cloud/CloudAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/context/ConfigurationPropertiesAutoConfiguration.html) |
| [CouchbaseAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/couchbase/CouchbaseAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/couchbase/CouchbaseAutoConfiguration.html) |
| [CouchbaseDataAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/couchbase/CouchbaseDataAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/couchbase/CouchbaseDataAutoConfiguration.html) |
| [CouchbaseRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/couchbase/CouchbaseRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/couchbase/CouchbaseRepositoriesAutoConfiguration.html) |
| [DataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.html) |
| [DataSourceTransactionManagerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceTransactionManagerAutoConfiguration.java)  [DeviceDelegatingViewResolverAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/DeviceDelegatingViewResolverAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jdbc/DataSourceTransactionManagerAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mobile/DeviceDelegatingViewResolverAutoConfiguration.html) |
| [DeviceResolverAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/DeviceResolverAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mobile/DeviceResolverAutoConfiguration.html) |
| [DispatcherServletAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/DispatcherServletAutoConfiguration.java)  [ElasticsearchAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/DispatcherServletAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchAutoConfiguration.html) |
| [ElasticsearchDataAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchDataAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchDataAutoConfiguration.html) |
| [ElasticsearchRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchRepositoriesAutoConfiguration.java)  [EmbeddedLdapAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/ldap/embedded/EmbeddedLdapAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchRepositoriesAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/ldap/embedded/EmbeddedLdapAutoConfiguration.html) |

|  |  |
| --- | --- |
| **Configuration Class** | **Links** |
| [EmbeddedMongoAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mongo/embedded/EmbeddedMongoAutoConfiguration.java)  [EmbeddedServletContainerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/EmbeddedServletContainerAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mongo/embedded/EmbeddedMongoAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/EmbeddedServletContainerAutoConfiguration.html) |
| [ErrorMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration.java)  [FacebookAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/FacebookAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/social/FacebookAutoConfiguration.html) |
| [FallbackWebSecurityAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/FallbackWebSecurityAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/security/FallbackWebSecurityAutoConfiguration.html) |
| [FlywayAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayAutoConfiguration.java)  [FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/flyway/FlywayAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.html) |
| [GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java)  [GsonAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/gson/GsonAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/gson/GsonAutoConfiguration.html) |
| [H2ConsoleAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/h2/H2ConsoleAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/h2/H2ConsoleAutoConfiguration.html) |
| [HazelcastAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hazelcast/HazelcastAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/hazelcast/HazelcastAutoConfiguration.html) |
| [HazelcastJpaDependencyAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hazelcast/HazelcastJpaDependencyAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/hazelcast/HazelcastJpaDependencyAutoConfiguration.html) |
| [HibernateJpaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.java)  [HttpEncodingAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/HttpEncodingAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/HttpEncodingAutoConfiguration.html) |
| [HttpMessageConvertersAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/HttpMessageConvertersAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/HttpMessageConvertersAutoConfiguration.html) |
| [HypermediaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hateoas/HypermediaAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/hateoas/HypermediaAutoConfiguration.html) |
| [IntegrationAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/integration/IntegrationAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/integration/IntegrationAutoConfiguration.html) |
| [JacksonAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jackson/JacksonAutoConfiguration.java)  [JdbcTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/JdbcTemplateAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jackson/JacksonAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jdbc/JdbcTemplateAutoConfiguration.html) |
| [JerseyAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jersey/JerseyAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jersey/JerseyAutoConfiguration.html) |
| [JestAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/elasticsearch/jest/JestAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/elasticsearch/jest/JestAutoConfiguration.html) |
| [JmsAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/JmsAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jms/JmsAutoConfiguration.html) |
| [JmxAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jmx/JmxAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jmx/JmxAutoConfiguration.html) |
| [JndiConnectionFactoryAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/JndiConnectionFactoryAutoConfiguration.java)  [JndiDataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/JndiDataSourceAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jms/JndiConnectionFactoryAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jdbc/JndiDataSourceAutoConfiguration.html) |
| [JooqAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jooq/JooqAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jooq/JooqAutoConfiguration.html) |
| [JpaRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/jpa/JpaRepositoriesAutoConfiguration.java)  [JtaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/transaction/jta/JtaAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/jpa/JpaRepositoriesAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/transaction/jta/JtaAutoConfiguration.html) |
| [KafkaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/kafka/KafkaAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/kafka/KafkaAutoConfiguration.html) |
| [LdapAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/ldap/LdapAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/ldap/LdapAutoConfiguration.html) |

|  |  |
| --- | --- |
| **Configuration Class** | **Links** |
| [LdapDataAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/ldap/LdapDataAutoConfiguration.java)  [LdapRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/ldap/LdapRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/ldap/LdapDataAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/ldap/LdapRepositoriesAutoConfiguration.html) |
| [LinkedInAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/LinkedInAutoConfiguration.java)  [LiquibaseAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/liquibase/LiquibaseAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/social/LinkedInAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/liquibase/LiquibaseAutoConfiguration.html) |
| [MailSenderAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mail/MailSenderAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mail/MailSenderAutoConfiguration.html) |
| [MailSenderValidatorAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mail/MailSenderValidatorAutoConfiguration.java)  [MessageSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/context/MessageSourceAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mail/MailSenderValidatorAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/context/MessageSourceAutoConfiguration.html) |
| [MongoAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mongo/MongoAutoConfiguration.java)  [MongoDataAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/mongo/MongoDataAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mongo/MongoAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/mongo/MongoDataAutoConfiguration.html) |
| [MongoRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/mongo/MongoRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/mongo/MongoRepositoriesAutoConfiguration.html) |
| [MultipartAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/MultipartAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/MultipartAutoConfiguration.html) |
| [MustacheAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mustache/MustacheAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mustache/MustacheAutoConfiguration.html) |
| [Neo4jDataAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/neo4j/Neo4jDataAutoConfiguration.java)  [Neo4jRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/neo4j/Neo4jRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/neo4j/Neo4jDataAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/neo4j/Neo4jRepositoriesAutoConfiguration.html) |
| [OAuth2AutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/OAuth2AutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/security/oauth2/OAuth2AutoConfiguration.html) |
| [PersistenceExceptionTranslationAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/dao/PersistenceExceptionTranslationAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/dao/PersistenceExceptionTranslationAutoConfiguration.html) |
| [ProjectInfoAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/info/ProjectInfoAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/info/ProjectInfoAutoConfiguration.html) |
| [PropertyPlaceholderAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/context/PropertyPlaceholderAutoConfiguration.java)  [RabbitAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/amqp/RabbitAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/context/PropertyPlaceholderAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/amqp/RabbitAutoConfiguration.html) |
| [ReactorAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/reactor/ReactorAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/reactor/ReactorAutoConfiguration.html) |
| [RedisAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/redis/RedisAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/redis/RedisAutoConfiguration.html) |
| [RedisRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/redis/RedisRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/redis/RedisRepositoriesAutoConfiguration.html) |
| [RepositoryRestMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/rest/RepositoryRestMvcAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/rest/RepositoryRestMvcAutoConfiguration.html) |
| [SecurityAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityAutoConfiguration.java)  [SecurityFilterAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityFilterAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/security/SecurityAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/security/SecurityFilterAutoConfiguration.html) |
| [SendGridAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/sendgrid/SendGridAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/sendgrid/SendGridAutoConfiguration.html) |
| [ServerPropertiesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerPropertiesAutoConfiguration.java)  [SessionAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/session/SessionAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/ServerPropertiesAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/session/SessionAutoConfiguration.html) |
| [SitePreferenceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/SitePreferenceAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/mobile/SitePreferenceAutoConfiguration.html) |
| [SocialWebAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/SocialWebAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/social/SocialWebAutoConfiguration.html) |

|  |  |
| --- | --- |
| **Configuration Class** | **Links** |
| [SolrAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/solr/SolrAutoConfiguration.java)  [SolrRepositoriesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/solr/SolrRepositoriesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/solr/SolrAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/solr/SolrRepositoriesAutoConfiguration.html) |
| [SpringApplicationAdminJmxAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/admin/SpringApplicationAdminJmxAutoConfiguration.java)  [SpringDataWebAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/web/SpringDataWebAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/admin/SpringApplicationAdminJmxAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/data/web/SpringDataWebAutoConfiguration.html) |
| [ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.html) |
| [TransactionAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/transaction/TransactionAutoConfiguration.java)  [TwitterAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/TwitterAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/transaction/TransactionAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/social/TwitterAutoConfiguration.html) |
| [ValidationAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/validation/ValidationAutoConfiguration.java)  [WebClientAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebClientAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/validation/ValidationAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/WebClientAutoConfiguration.html) |
| [WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.html) |
| [WebServicesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/webservices/WebServicesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/webservices/WebServicesAutoConfiguration.html) |
| [WebSocketAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/websocket/WebSocketAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/websocket/WebSocketAutoConfiguration.html) |
| [WebSocketMessagingAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/websocket/WebSocketMessagingAutoConfiguration.java)  [XADataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/XADataSourceAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/websocket/WebSocketMessagingAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/autoconfigure/jdbc/XADataSourceAutoConfiguration.html) |

## From the “spring-boot-actuator” module

The following auto-configuration classes are from the spring-boot-actuator module:

|  |  |
| --- | --- |
| **Configuration Class**  [AuditAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/AuditAutoConfiguration.java) | **Links**  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/AuditAutoConfiguration.html) |
| [CacheStatisticsAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/CacheStatisticsAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/CacheStatisticsAutoConfiguration.html) |
| [CloudFoundryActuatorAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/cloudfoundry/CloudFoundryActuatorAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/cloudfoundry/CloudFoundryActuatorAutoConfiguration.html) |
| [CrshAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/CrshAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/CrshAutoConfiguration.html) |
| [EndpointAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/EndpointAutoConfiguration.html) |
| [EndpointMBeanExportAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointMBeanExportAutoConfiguration.java)  [EndpointWebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointWebMvcAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/EndpointMBeanExportAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/EndpointWebMvcAutoConfiguration.html) |
| [HealthIndicatorAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/HealthIndicatorAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/HealthIndicatorAutoConfiguration.html) |
| [InfoContributorAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/InfoContributorAutoConfiguration.java)  [JolokiaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/JolokiaAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/InfoContributorAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/JolokiaAutoConfiguration.html) |
| [ManagementServerPropertiesAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ManagementServerPropertiesAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/ManagementServerPropertiesAutoConfiguration.html) |
| [ManagementWebSecurityAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ManagementWebSecurityAutoConfiguration.java)  [MetricExportAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/MetricExportAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/ManagementWebSecurityAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/MetricExportAutoConfiguration.html) |

|  |  |
| --- | --- |
| **Configuration Class** | **Links** |
| [MetricFilterAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/MetricFilterAutoConfiguration.java)  [MetricRepositoryAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/MetricRepositoryAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/MetricFilterAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/MetricRepositoryAutoConfiguration.html) |
| [MetricsChannelAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/MetricsChannelAutoConfiguration.java)  [MetricsDropwizardAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/MetricsDropwizardAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/MetricsChannelAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/MetricsDropwizardAutoConfiguration.html) |
| [PublicMetricsAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/PublicMetricsAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/PublicMetricsAutoConfiguration.html) |
| [TraceRepositoryAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/TraceRepositoryAutoConfiguration.java)  [TraceWebFilterAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.8.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/TraceWebFilterAutoConfiguration.java) | [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/TraceRepositoryAutoConfiguration.html)  [javadoc](http://docs.spring.io/spring-boot/docs/1.5.8.RELEASE/api/org/springframework/boot/actuate/autoconfigure/TraceWebFilterAutoConfiguration.html) |

**Appendix D. Test auto-configuration annotations**

Here is a table of the various @…Test annotations that can be used to test slices of your application and the auto-configuration that they import by default:

|  |  |
| --- | --- |
| **Test slice** | **Imported auto-configuration** |
| @DataJpaTest | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.test.autocon |
|  | org.springframework.boot.test.autocon |
| @DataMongoTest | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
| @JdbcTest | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.test.autocon |
| @JsonTest | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.test.autocon |
| @RestClientTest | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.autoconfigur |
|  | org.springframework.boot.test.autocon |
|  | org.springframework.boot.test.autocon |
| @WebMvcTest org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur | |

e.cache.Cach e.data.jpa.J e.flyway.Fly e.jdbc.DataS e.jdbc.DataS e.jdbc.JdbcT e.liquibase. e.orm.jpa.Hi e.transactio figure.jdbc. figure.orm.j

e.cache.Cach e.data.mongo e.data.mongo e.mongo.Mong e.mongo.embe

e.cache.Cach e.flyway.Fly e.jdbc.DataS e.jdbc.DataS e.jdbc.JdbcT e.liquibase. e.transactio figure.jdbc.

e.cache.Cach e.gson.GsonA e.jackson.Ja figure.json.

e.cache.Cach e.gson.GsonA e.jackson.Ja e.web.HttpMe e.web.WebCli figure.web.c figure.web.c

e.cache.Cach e.context.Me

e.freemarker e.groovy.tem e.gson.GsonA e.hateoas.Hy e.jackson.Ja e.mustache.M e.thymeleaf. e.validation e.web.ErrorM e.web.HttpMe e.web.Server e.web.WebMvc figure.web.s figure.web.s figure.web.s figure.web.s

|  |
| --- |
| **Test slice Imported auto-configuration** |
| org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.autoconfigur org.springframework.boot.test.autocon org.springframework.boot.test.autocon org.springframework.boot.test.autocon org.springframework.boot.test.autocon |

**Appendix E. The executable jar format**

The spring-boot-loader modules allows Spring Boot to support executable jar and war files. If you’re using the Maven or Gradle plugin, executable jars are automatically generated and you generally won’t 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 section provides some background.

## Nested JARs

Java does not provide any standard way to load nested jar files (i.e. jar files that are themselves contained within a jar). This can be problematic if you are looking to distribute a self-contained application that you can just run from the command line without unpacking.

To solve this problem, many developers use “shaded” jars. A shaded jar simply packages all classes, from all jars, into a single 'uber jar'. The problem with shaded jars is that it becomes hard to see which libraries you are actually using in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars. Spring Boot takes a different approach and allows you to actually nest jars directly.

### The executable jar file structure

Spring Boot Loader compatible jar files should be structured in the following way:

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

Application classes should be placed in a nested BOOT-INF/classes directory. Dependencies should be placed in a nested BOOT-INF/lib directory.

### The executable war file structure

Spring Boot Loader compatible war files should be structured in the following way:

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

Dependencies should be placed in a nested WEB-INF/lib directory. Any dependencies that are required when running embedded but are not required when deploying to a traditional web container should be placed in WEB-INF/lib-provided.

## Spring Boot’s “JarFile” class

The core class used to support loading nested jars is org.springframework.boot.loader.jar.JarFile. It allows you to load jar content from a standard jar file, or from nested child jar data. When first loaded, the location of each JarEntry is mapped to a physical file offset of the outer jar:

myapp.jar

+-------------------+-------------------------+

| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |

|+-----------------+||+-----------+----------+|

|| A.class ||| B.class | C.class ||

|+-----------------+||+-----------+----------+|

+-------------------+-------------------------+

^ ^ ^

0063 3452 3980

The example above shows how A.class can be found in /BOOT-INF/classes in myapp.jar position 0063. B.class from the nested jar can actually be found in myapp.jar position 3452 and C.class is at position 3980.

Armed with this information, we can load specific nested entries by simply seeking to the appropriate part of the outer jar. We don’t need to unpack the archive and we don’t need to read all entry data into memory.

### Compatibility with the standard Java “JarFile”

Spring Boot Loader strives to remain compatible with existing code and libraries. org.springframework.boot.loader.jar.JarFile extends from java.util.jar.JarFile and should work as a drop-in replacement. The getURL() method will return a URL that opens a java.net.JarURLConnection compatible connection and can be used with Java’s URLClassLoader.

## Launching executable jars

The org.springframework.boot.loader.Launcher class is a special bootstrap class that is used as an executable jars main entry point. It is the actual Main-Class in your jar file and it’s used to setup an appropriate URLClassLoader and ultimately call your main() method.

There are 3 launcher subclasses (JarLauncher, WarLauncher and PropertiesLauncher). Their purpose is to load resources (.class files etc.) from nested jar files or war files in directories (as opposed to explicitly on the classpath). In the case of JarLauncher and WarLauncher the nested paths are fixed. JarLauncher looks in BOOT-INF/lib/ and WarLauncher looks in WEB-INF/ lib/ and WEB-INF/lib-provided/ so you just add extra jars in those locations if you want more. The PropertiesLauncher looks in BOOT-INF/lib/ in your application archive by default, but you can add additional locations by setting an environment variable LOADER\_PATH or loader.path in loader.properties (comma-separated list of directories, archives, or directories within archives).

### Launcher manifest

You need to specify an appropriate Launcher as the Main-Class attribute of META-INF/ MANIFEST.MF. The actual class that you want to launch (i.e. the class that you wrote that contains a main method) should be specified in the Start-Class attribute.

For example, here is a typical MANIFEST.MF for an executable jar file:

Main-Class: org.springframework.boot.loader.JarLauncher Start-Class: com.mycompany.project.MyApplication

For a war file, it would be:

Main-Class: org.springframework.boot.loader.WarLauncher Start-Class: com.mycompany.project.MyApplication

**Note**

You do not need to specify Class-Path entries in your manifest file, the classpath will be deduced from the nested jars.

### Exploded archives

Certain PaaS implementations may choose to unpack archives before they run. For example, Cloud Foundry operates in this way. You can run an unpacked archive by simply starting the appropriate launcher:

$ unzip -q myapp.jar

$ java org.springframework.boot.loader.JarLauncher

## PropertiesLauncher Features

PropertiesLauncher has a few special features that can be enabled with external properties (System properties, environment variables, manifest entries or loader.properties).

**Note**

PropertiesLauncher supports loading properties from loader.properties and also (for historic reasons) application.properties. We recommend using loader.properties exclusively, as support for application.properties is deprecated and may be removed in the future.

|  |  |
| --- | --- |
| **Key** | **Purpose** |
| loader.path  loader.home | Comma-separated Classpath, e.g. lib,  ${HOME}/app/lib. Earlier entries take precedence, just like a regular -classpath on the javac command line.  Used to resolve relative paths in  loader.path. E.g. loader.path=lib then  ${loader.home}/lib is a classpath location (along with all jar files in that directory). Also used to locate a loader.properties file.  Example /opt/app (defaults to ${user.dir}). |
| loader.args | Default arguments for the main method (space |
|  | separated) |
| loader.main | Name of main class to launch, e.g. |
|  | com.app.Application. |
| loader.config.name | Name of properties file, e.g. launcher (defaults to loader). |
| loader.config.location | Path to properties file, e.g. classpath:loader.properties (defaults to loader.properties). |
| loader.system | Boolean flag to indicate that all properties should be added to System properties (defaults to false) |

When specified as environment variables or manifest entries, the following names should be used:

|  |  |  |
| --- | --- | --- |
| **Key** | **Manifest entry** | **Environment variable** |
| loader.path  loader.home | Loader-Path  Loader-Home | LOADER\_PATH  LOADER\_HOME |
| loader.args | Loader-Args | LOADER\_ARGS |
| loader.main  loader.config.location | Start-Class  Loader-Config-Location | LOADER\_MAIN  LOADER\_CONFIG\_LOCATION |
| loader.system | Loader-System | LOADER\_SYSTEM |

**Tip**

Build plugins automatically move the Main-Class attribute to Start-Class when the fat jar is built. If you are using that, specify the name of the class to launch using the Main-Class attribute and leave out Start-Class.

* loader.properties are searched for in loader.home then in the root of the classpath, then in

classpath:/BOOT-INF/classes. The first location that exists is used.

* loader.home is only the directory location of an additional properties file (overriding the default) as long as loader.config.location is not specified.
* loader.path can contain directories (scanned recursively for jar and zip files), archive paths, a directory within an archive that is scanned for jar files (for example, dependencies.jar!/lib), or wildcard patterns (for the default JVM behavior). Archive paths can be relative to loader.home, or anywhere in the file system with a jar:file: prefix.
* loader.path (if empty) defaults to BOOT-INF/lib (meaning a local directory or a nested one if running from an archive). Because of this PropertiesLauncher behaves the same as JarLauncher when no additional configuration is provided.
* loader.path can not be used to configure the location of loader.properties (the classpath used to search for the latter is the JVM classpath when PropertiesLauncher is launched).
* Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.
* The search order for properties (where it makes sense to look in more than one place) is env vars, system properties, loader.properties, exploded archive manifest, archive manifest.

## Executable jar restrictions

There are a number of restrictions that you need to consider when working with a Spring Boot Loader packaged application.

### Zip entry compression

The ZipEntry for a nested jar must be saved using the ZipEntry.STORED method. This is required so that we can seek directly to individual content within the nested jar. The content of the nested jar file itself can still be compressed, as can any other entries in the outer jar.

### System ClassLoader

Launched applications should use Thread.getContextClassLoader() when loading classes (most libraries and frameworks will do this by default). Trying to load nested jar classes via ClassLoader.getSystemClassLoader() will fail. Please be aware that java.util.Logging always uses the system classloader, for this reason you should consider a different logging implementation.

## Alternative single jar solutions

If the above restrictions mean that you cannot use Spring Boot Loader the following alternatives could be considered:

* [Maven Shade Plugin](http://maven.apache.org/plugins/maven-shade-plugin/)
* [JarClassLoader](http://www.jdotsoft.com/JarClassLoader.php)
* [OneJar](http://one-jar.sourceforge.net/)

**Appendix F. Dependency versions**

The table below provides details of all of the dependency versions that are provided by Spring Boot in its CLI, Maven dependency management and Gradle plugin. When you declare a dependency on one of these artifacts without declaring a version the version that is listed in the table will be used.

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| antlr | antlr | 2.7.7 |
| ch.qos.logback | logback-access | 1.1.11 |
| ch.qos.logback | logback-classic | 1.1.11 |
| ch.qos.logback | logback-core | 1.1.11 |
| com.atomikos | transactions-jdbc | 3.9.3 |
| com.atomikos | transactions-jms | 3.9.3 |
| com.atomikos | transactions-jta | 3.9.3 |
| com.couchbase.client | couchbase-spring-cache | 2.1.0 |
| com.couchbase.client | java-client | 2.3.7 |
| com.datastax.cassandra | cassandra-driver-core | 3.1.4 |
| com.datastax.cassandra | cassandra-driver- mapping | 3.1.4 |
| com.fasterxml | classmate | 1.3.4 |

com.fasterxml.jackson.corejackson-annotations 2.8.0

com.fasterxml.jackson.corejackson-core 2.8.10

com.fasterxml.jackson.corejackson-databind 2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-avro 2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-cbor 2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-csv 2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-ion 2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-

properties

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-

protobuf

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-

smile

2.8.10

2.8.10

2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-xml 2.8.10

com.fasterxml.jackson.datajfaocrkmsaotn-dataformat-yaml 2.8.10

**Group ID Artifact ID Version**

com.fasterxml.jackson.datajtaycpkeson-datatype-guava 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-

hibernate3

com.fasterxml.jackson.datajtaycpkeson-datatype-

hibernate4

com.fasterxml.jackson.datajtaycpkeson-datatype-

hibernate5

2.8.10

2.8.10

2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-hppc 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-jaxrs 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-jdk8 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-joda 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-json-

org

2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-jsr310 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-jsr353 2.8.10

com.fasterxml.jackson.datajtaycpkeson-datatype-

pcollections

2.8.10

com.fasterxml.jackson.jaxrjsackson-jaxrs-base 2.8.10

com.fasterxml.jackson.jaxrjsackson-jaxrs-cbor-

provider

com.fasterxml.jackson.jaxrjsackson-jaxrs-json-

provider

com.fasterxml.jackson.jaxrjsackson-jaxrs-smile-

provider

com.fasterxml.jackson.jaxrjsackson-jaxrs-xml-

provider

com.fasterxml.jackson.jaxrjsackson-jaxrs-yaml-

provider

2.8.10

2.8.10

2.8.10

2.8.10

2.8.10

com.fasterxml.jackson.jr jackson-jr-all 2.8.10

com.fasterxml.jackson.jr jackson-jr-objects 2.8.10

com.fasterxml.jackson.jr jackson-jr-retrofit2 2.8.10

com.fasterxml.jackson.jr jackson-jr-stree 2.8.10

com.fasterxml.jackson.modujlaeckson-module-

afterburner

2.8.10

#### Group ID Artifact ID Version

com.fasterxml.jackson.modujlaeckson-module-guice 2.8.10

com.fasterxml.jackson.modujlaeckson-module-jaxb-

annotations

com.fasterxml.jackson.modujlaeckson-module-

jsonSchema

2.8.10

2.8.10

com.fasterxml.jackson.modujlaeckson-module-kotlin 2.8.10

com.fasterxml.jackson.modujlaeckson-module-mrbean 2.8.10

com.fasterxml.jackson.modujlaeckson-module-osgi 2.8.10

com.fasterxml.jackson.modujlaeckson-module-

parameter-names

com.fasterxml.jackson.modujlaeckson-module-

paranamer

com.fasterxml.jackson.modujlaeckson-module-

scala\_2.10

com.fasterxml.jackson.modujlaeckson-module-

scala\_2.11

com.fasterxml.jackson.modujlaeckson-module-

scala\_2.12

2.8.10

2.8.10

2.8.10

2.8.10

2.8.10

com.gemstone.gemfire gemfire 8.2.7

com.github.ben- manes.caffeine

caffeine 2.3.5

com.github.mxab.thymeleaf.tehxytmrealseaf-extras-data-

attribute

1.3

com.google.appengine appengine-api-1.0-sdk 1.9.58

com.google.code.gson gson 2.8.2

com.googlecode.json- simple

json-simple 1.1.1

com.h2database h2 1.4.196

com.hazelcast hazelcast 3.7.8

com.hazelcast hazelcast-client 3.7.8

com.hazelcast hazelcast-hibernate4 3.7.1

com.hazelcast hazelcast-hibernate5 1.1.3

com.hazelcast hazelcast-spring 3.7.8

com.jayway.jsonpath json-path 2.2.0

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| com.jayway.jsonpath  com.microsoft.sqlserver | json-path-assert  mssql-jdbc | 2.2.0  6.1.0.jre7 |
| com.querydsl  com.querydsl | querydsl-apt  querydsl-collections | 4.1.4  4.1.4 |
| com.querydsl | querydsl-core | 4.1.4 |
| com.querydsl  com.querydsl | querydsl-jpa  querydsl-mongodb | 4.1.4  4.1.4 |
| com.samskivert  com.sendgrid | jmustache  sendgrid-java | 1.13  2.2.2 |
| com.sun.mail | javax.mail | 1.5.6 |
| com.timgroup | java-statsd-client | 3.1.0 |
| com.unboundid | unboundid-ldapsdk | 3.2.1 |
| com.zaxxer  com.zaxxer | HikariCP  HikariCP-java6 | 2.5.1  2.3.13 |
| com.zaxxer | HikariCP-java7 | 2.4.13 |
| commons-beanutils | commons-beanutils | 1.9.3 |
| commons-codec | commons-codec | 1.10 |
| commons-collections  commons-dbcp | commons-collections  commons-dbcp | 3.2.2  1.4 |
| commons-digester | commons-digester | 2.1 |
| commons-pool | commons-pool | 1.6 |
| de.flapdoodle.embed | de.flapdoodle.embed.mongo1.50.5 | |
| dom4j | dom4j | 1.6.1 |
| io.dropwizard.metrics  io.dropwizard.metrics | metrics-annotation  metrics-core | 3.1.5  3.1.5 |
| io.dropwizard.metrics | metrics-ehcache | 3.1.5 |
| io.dropwizard.metrics  io.dropwizard.metrics | metrics-ganglia  metrics-graphite | 3.1.5  3.1.5 |
| io.dropwizard.metrics | metrics-healthchecks | 3.1.5 |
| io.dropwizard.metrics | metrics-httpasyncclient | 3.1.5 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| io.dropwizard.metrics  io.dropwizard.metrics | metrics-jdbi  metrics-jersey | 3.1.5  3.1.5 |
| io.dropwizard.metrics  io.dropwizard.metrics | metrics-jersey2  metrics-jetty8 | 3.1.5  3.1.5 |
| io.dropwizard.metrics | metrics-jetty9 | 3.1.5 |
| io.dropwizard.metrics  io.dropwizard.metrics | metrics-jetty9-legacy  metrics-json | 3.1.5  3.1.5 |
| io.dropwizard.metrics  io.dropwizard.metrics | metrics-jvm  metrics-log4j | 3.1.5  3.1.5 |
| io.dropwizard.metrics | metrics-log4j2 | 3.1.5 |
| io.dropwizard.metrics | metrics-logback | 3.1.5 |
| io.dropwizard.metrics | metrics-servlet | 3.1.5 |
| io.dropwizard.metrics  io.projectreactor | metrics-servlets  reactor-bus | 3.1.5  2.0.8.RELEASE |
| io.projectreactor | reactor-core | 2.0.8.RELEASE |
| io.projectreactor | reactor-groovy | 2.0.8.RELEASE |
| io.projectreactor | reactor-groovy- extensions | 2.0.8.RELEASE |
| io.projectreactor  io.projectreactor | reactor-logback  reactor-net | 2.0.8.RELEASE  2.0.8.RELEASE |
| io.projectreactor | reactor-stream | 2.0.8.RELEASE |
| io.projectreactor.spring | reactor-spring-context | 2.0.7.RELEASE |
| io.projectreactor.spring io.projectreactor.spring | reactor-spring-core  reactor-spring- messaging | 2.0.7.RELEASE  2.0.7.RELEASE |
| io.projectreactor.spring  io.searchbox | reactor-spring-webmvc  jest | 2.0.7.RELEASE  2.0.4 |
| io.undertow | undertow-core | 1.4.20.Final |
| io.undertow  io.undertow | undertow-servlet  undertow-websockets-jsr | 1.4.20.Final  1.4.20.Final |
| javax.cache | cache-api | 1.0.0 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| javax.jms  javax.mail | jms-api  javax.mail-api | 1.1-rev-1  1.5.6 |
| javax.servlet  javax.servlet | javax.servlet-api  jstl | 3.1.0  1.2 |
| javax.transaction | javax.transaction-api | 1.2 |
| javax.validation  jaxen | validation-api  jaxen | 1.1.0.Final  1.1.6 |
| joda-time  junit | joda-time  junit | 2.9.9  4.12 |
| mysql | mysql-connector-java | 5.1.44 |
| net.java.dev.jna | jna | 4.2.2 |
| net.java.dev.jna | jna-platform | 4.2.2 |
| net.sf.ehcache  net.sourceforge.htmlunit | ehcache  htmlunit | 2.10.4  2.21 |
| net.sourceforge.jtds | jtds | 1.3.1 |
| net.sourceforge.nekohtml | nekohtml | 1.9.22 |
| nz.net.ultraq.thymeleaf | thymeleaf-layout- dialect | 1.4.0 |
| org.apache.activemq  org.apache.activemq | activemq-amqp  activemq-blueprint | 5.14.5  5.14.5 |
| org.apache.activemq | activemq-broker | 5.14.5 |
| org.apache.activemq | activemq-camel | 5.14.5 |
| org.apache.activemq  org.apache.activemq | activemq-client  activemq-console | 5.14.5  5.14.5 |
| org.apache.activemq  org.apache.activemq | activemq-http  activemq-jaas | 5.14.5  5.14.5 |
| org.apache.activemq | activemq-jdbc-store | 5.14.5 |
| org.apache.activemq  org.apache.activemq | activemq-jms-pool  activemq-kahadb-store | 5.14.5  5.14.5 |
| org.apache.activemq | activemq-karaf | 5.14.5 |
| org.apache.activemq | activemq-leveldb-store | 5.14.5 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.apache.activemq | activemq-log4j-appender | 5.14.5 |
| org.apache.activemq | activemq-mqtt | 5.14.5 |
| org.apache.activemq | activemq-openwire- | 5.14.5 |
|  | generator |  |
| org.apache.activemq | activemq-openwire- | 5.14.5 |
|  | legacy |  |
| org.apache.activemq | activemq-osgi | 5.14.5 |
| org.apache.activemq | activemq-partition | 5.14.5 |
| org.apache.activemq | activemq-pool | 5.14.5 |
| org.apache.activemq | activemq-ra | 5.14.5 |
| org.apache.activemq | activemq-run | 5.14.5 |
| org.apache.activemq | activemq-runtime-config | 5.14.5 |
| org.apache.activemq | activemq-shiro | 5.14.5 |
| org.apache.activemq | activemq-spring | 5.14.5 |
| org.apache.activemq | activemq-stomp | 5.14.5 |
| org.apache.activemq | activemq-web | 5.14.5 |
| org.apache.activemq | artemis-amqp-protocol | 1.5.5 |
| org.apache.activemq | artemis-commons | 1.5.5 |
| org.apache.activemq | artemis-core-client | 1.5.5 |
| org.apache.activemq | artemis-jms-client | 1.5.5 |
| org.apache.activemq | artemis-jms-server | 1.5.5 |
| org.apache.activemq | artemis-journal | 1.5.5 |
| org.apache.activemq | artemis-native | 1.5.5 |
| org.apache.activemq | artemis-selector | 1.5.5 |
| org.apache.activemq | artemis-server | 1.5.5 |
| org.apache.activemq | artemis-service- | 1.5.5 |
|  | extensions |  |
| org.apache.commons | commons-dbcp2 | 2.1.1 |
| org.apache.commons | commons-pool2 | 2.4.2 |
| org.apache.derby  org.apache.httpcomponent | derby 10.13.1.1  shttpasyncclient 4.1.3 | |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.apache.httpcomponent  org.apache.httpcomponent | shttpclient 4.5.3  shttpcore 4.4.8 | |
| org.apache.httpcomponent | shttpmime  log4j-1.2-api | 4.5.3 |
| org.apache.logging.log4j | 2.7 |
| org.apache.logging.log4j | log4j-api | 2.7 |
| org.apache.logging.log4j | log4j-api-scala\_2.10 | 2.7 |
| org.apache.logging.log4j | log4j-api-scala\_2.11 | 2.7 |
| org.apache.logging.log4j | log4j-core | 2.7 |
| org.apache.logging.log4j | log4j-flume-ng | 2.7 |
| org.apache.logging.log4j | log4j-iostreams | 2.7 |
| org.apache.logging.log4j | log4j-jcl | 2.7 |
| org.apache.logging.log4j | log4j-jmx-gui | 2.7 |
| org.apache.logging.log4j | log4j-jul | 2.7 |
| org.apache.logging.log4j | log4j-liquibase | 2.7 |
| org.apache.logging.log4j | log4j-nosql | 2.7 |
| org.apache.logging.log4j | log4j-slf4j-impl | 2.7 |
| org.apache.logging.log4j | log4j-taglib | 2.7 |
| org.apache.logging.log4j | log4j-web | 2.7 |
| org.apache.solr | solr-analysis-extras | 5.5.4 |
| org.apache.solr | solr-analytics | 5.5.4 |
| org.apache.solr | solr-cell | 5.5.4 |
| org.apache.solr | solr-clustering | 5.5.4 |
| org.apache.solr | solr-core | 5.5.4 |
| org.apache.solr | solr-dataimporthandler | 5.5.4 |
| org.apache.solr | solr-dataimporthandler- | 5.5.4 |
|  | extras |  |
| org.apache.solr | solr-langid | 5.5.4 |
| org.apache.solr | solr-map-reduce | 5.5.4 |
| org.apache.solr | solr-morphlines-cell | 5.5.4 |
| org.apache.solr | solr-morphlines-core | 5.5.4 |
| org.apache.solr | solr-solrj | 5.5.4 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.apache.solr | solr-test-framework | 5.5.4 |
| org.apache.solr | solr-uima | 5.5.4 |
| org.apache.solr | solr-velocity | 5.5.4 |
| org.apache.tomcat | tomcat-annotations-api | 8.5.23 |
| org.apache.tomcat | tomcat-jdbc | 8.5.23 |
| org.apache.tomcat | tomcat-jsp-api | 8.5.23 |
| org.apache.tomcat.embed | tomcat-embed-core | 8.5.23 |
| org.apache.tomcat.embed | tomcat-embed-el | 8.5.23 |
| org.apache.tomcat.embed | tomcat-embed-jasper | 8.5.23 |
| org.apache.tomcat.embed | tomcat-embed-websocket | 8.5.23 |
| org.aspectj | aspectjrt | 1.8.11 |
| org.aspectj | aspectjtools | 1.8.11 |
| org.aspectj | aspectjweaver | 1.8.11 |
| org.assertj | assertj-core | 2.6.0 |
| org.codehaus.btm | btm | 2.1.4 |
| org.codehaus.groovy | groovy | 2.4.12 |
| org.codehaus.groovy | groovy-all | 2.4.12 |
| org.codehaus.groovy | groovy-ant | 2.4.12 |
| org.codehaus.groovy | groovy-bsf | 2.4.12 |
| org.codehaus.groovy | groovy-console | 2.4.12 |
| org.codehaus.groovy | groovy-docgenerator | 2.4.12 |
| org.codehaus.groovy | groovy-groovydoc | 2.4.12 |
| org.codehaus.groovy | groovy-groovysh | 2.4.12 |
| org.codehaus.groovy | groovy-jmx | 2.4.12 |
| org.codehaus.groovy | groovy-json | 2.4.12 |
| org.codehaus.groovy | groovy-jsr223 | 2.4.12 |
| org.codehaus.groovy | groovy-nio | 2.4.12 |
| org.codehaus.groovy | groovy-servlet | 2.4.12 |
| org.codehaus.groovy | groovy-sql | 2.4.12 |
| org.codehaus.groovy | groovy-swing | 2.4.12 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.codehaus.groovy | groovy-templates | 2.4.12 |
| org.codehaus.groovy | groovy-test | 2.4.12 |
| org.codehaus.groovy | groovy-testng | 2.4.12 |
| org.codehaus.groovy | groovy-xml | 2.4.12 |
| org.codehaus.janino | janino | 2.7.8 |
| org.crashub | crash.cli | 1.3.2 |
| org.crashub | crash.connectors.ssh | 1.3.2 |
| org.crashub | crash.connectors.telnet | 1.3.2 |
| org.crashub | crash.embed.spring | 1.3.2 |
| org.crashub | crash.plugins.cron | 1.3.2 |
| org.crashub | crash.plugins.mail | 1.3.2 |
| org.crashub | crash.shell | 1.3.2 |
| org.eclipse.jetty | apache-jsp | 9.4.7.v20170914 |
| org.eclipse.jetty | apache-jstl | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-alpn-client | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-alpn-java-client | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-alpn-java-server | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-alpn-server | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-annotations | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-ant | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-client | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-continuation | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-deploy | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-hazelcast | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-http | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-http-spi | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-infinispan | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-io | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-jaas | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-jaspi | 9.4.7.v20170914 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.eclipse.jetty | jetty-jmx | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-jndi | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-nosql | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-plus | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-proxy | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-quickstart | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-rewrite | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-runner | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-security | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-server | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-servlet | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-servlets | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-spring | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-start | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-unixsocket | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-util | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-util-ajax | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-webapp | 9.4.7.v20170914 |
| org.eclipse.jetty | jetty-xml | 9.4.7.v20170914 |
| org.eclipse.jetty.cdi | cdi-core | 9.4.7.v20170914 |
| org.eclipse.jetty.cdi | cdi-servlet | 9.4.7.v20170914 |
| org.eclipse.jetty.fcgi | fcgi-client | 9.4.7.v20170914 |
| org.eclipse.jetty.fcgi | fcgi-server | 9.4.7.v20170914 |
| org.eclipse.jetty.gcloud | jetty-gcloud-session- | 9.4.7.v20170914 |
|  | manager |  |
| org.eclipse.jetty.http2 | http2-client | 9.4.7.v20170914 |
| org.eclipse.jetty.http2 | http2-common | 9.4.7.v20170914 |
| org.eclipse.jetty.http2 | http2-hpack | 9.4.7.v20170914 |
| org.eclipse.jetty.http2 | http2-http-client- | 9.4.7.v20170914 |
|  | transport |  |
| org.eclipse.jetty.http2 | http2-server | 9.4.7.v20170914 |

**Group ID Artifact ID Version**

org.eclipse.jetty.memcachejdetty-memcached-

sessions

9.4.7.v20170914

org.eclipse.jetty.orbit javax.servlet.jsp 2.2.0.v201112011158 org.eclipse.jetty.osgi jetty-httpservice 9.4.7.v20170914 org.eclipse.jetty.osgi jetty-osgi-boot 9.4.7.v20170914 org.eclipse.jetty.osgi jetty-osgi-boot-jsp 9.4.7.v20170914 org.eclipse.jetty.osgi jetty-osgi-boot-warurl 9.4.7.v20170914

org.eclipse.jetty.websockejtavax-websocket-client-

impl

org.eclipse.jetty.websockejtavax-websocket-server-

impl

9.4.7.v20170914

9.4.7.v20170914

org.eclipse.jetty.websockewtebsocket-api 9.4.7.v20170914 org.eclipse.jetty.websockewtebsocket-client 9.4.7.v20170914 org.eclipse.jetty.websockewtebsocket-common 9.4.7.v20170914 org.eclipse.jetty.websockewtebsocket-server 9.4.7.v20170914 org.eclipse.jetty.websockewtebsocket-servlet 9.4.7.v20170914 org.ehcache ehcache 3.2.3

org.ehcache ehcache-clustered 3.2.3

org.ehcache ehcache-transactions 3.2.3

org.elasticsearch elasticsearch 2.4.6

org.firebirdsql.jdbc jaybird-jdk16 2.2.13

org.firebirdsql.jdbc jaybird-jdk17 2.2.13

org.firebirdsql.jdbc jaybird-jdk18 2.2.13

org.flywaydb flyway-core 3.2.1

org.freemarker freemarker 2.3.26-incubating

org.glassfish javax.el 3.0.0

org.glassfish.jersey.bundljer.sreeyp-agcukaavgaed org.glassfish.jersey.contajienresresy-container-

servlet

org.glassfish.jersey.contajienresresy-container-

servlet-core

2.25.1

2.25.1

2.25.1

org.glassfish.jersey.corejersey-client 2.25.1

#### Group ID Artifact ID Version

org.glassfish.jersey.corejersey-common 2.25.1

org.glassfish.jersey.corejersey-server 2.25.1

org.glassfish.jersey.ext jersey-bean-validation 2.25.1

org.glassfish.jersey.ext jersey-entity-filtering 2.25.1

org.glassfish.jersey.ext jersey-spring3 2.25.1

org.glassfish.jersey.mediajersey-media-jaxb 2.25.1

org.glassfish.jersey.mediajersey-media-json-

jackson

2.25.1

org.glassfish.jersey.mediajersey-media-multipart 2.25.1

org.hamcrest hamcrest-core 1.3

org.hamcrest hamcrest-library 1.3

org.hibernate hibernate-core 5.0.12.Final

org.hibernate hibernate-ehcache 5.0.12.Final

org.hibernate hibernate-entitymanager 5.0.12.Final

org.hibernate hibernate-envers 5.0.12.Final

org.hibernate hibernate-java8 5.0.12.Final

org.hibernate hibernate-jpamodelgen 5.0.12.Final

org.hibernate hibernate-validator 5.3.5.Final

org.hibernate hibernate-validator- annotation-processor

5.3.5.Final

org.hsqldb hsqldb 2.3.5

org.infinispan infinispan-jcache 8.2.8.Final

org.infinispan infinispan-spring4- common

org.infinispan infinispan-spring4- embedded

8.2.8.Final

8.2.8.Final

org.javassist javassist 3.21.0-GA

org.jboss jboss-transaction-spi 7.6.0.Final

org.jboss.logging jboss-logging 3.3.1.Final

org.jboss.narayana.jta jdbc 5.5.30.Final

org.jboss.narayana.jta jms 5.5.30.Final

org.jboss.narayana.jta jta 5.5.30.Final

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.jboss.narayana.jts  org.jdom | narayana-jts- integration  jdom2 | 5.5.30.Final  2.0.6 |
| org.jolokia  org.jooq | jolokia-core  jooq | 1.3.7  3.9.6 |
| org.jooq | jooq-codegen | 3.9.6 |
| org.jooq  org.json | jooq-meta  json | 3.9.6  20140107 |
| org.liquibase  org.mariadb.jdbc | liquibase-core  mariadb-java-client | 3.5.3  1.5.9 |
| org.mockito | mockito-core | 1.10.19 |
| org.mongodb | mongodb-driver | 3.4.3 |
| org.mongodb | mongo-java-driver | 3.4.3 |
| org.mortbay.jasper  org.neo4j | apache-el  neo4j-ogm-api | 8.0.33  2.1.5 |
| org.neo4j | neo4j-ogm-compiler | 2.1.5 |
| org.neo4j | neo4j-ogm-core | 2.1.5 |
| org.neo4j | neo4j-ogm-http-driver | 2.1.5 |
| org.postgresql  org.projectlombok | postgresql  lombok | 9.4.1212.jre7  1.16.18 |
| org.seleniumhq.selenium | htmlunit-driver | 2.21 |
| org.seleniumhq.selenium | selenium-api | 2.53.1 |
| org.seleniumhq.selenium  org.seleniumhq.selenium | selenium-chrome-driver  selenium-firefox-driver | 2.53.1  2.53.1 |
| org.seleniumhq.selenium  org.seleniumhq.selenium | selenium-ie-driver  selenium-java | 2.53.1  2.53.1 |
| org.seleniumhq.selenium | selenium-remote-driver | 2.53.1 |
| org.seleniumhq.selenium  org.seleniumhq.selenium | selenium-safari-driver  selenium-support | 2.53.1  2.53.1 |
| org.skyscreamer | jsonassert | 1.4.0 |
| org.slf4j | jcl-over-slf4j | 1.7.25 |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.slf4j  org.slf4j | jul-to-slf4j  log4j-over-slf4j | 1.7.25  1.7.25 |
| org.slf4j  org.slf4j | slf4j-api  slf4j-ext | 1.7.25  1.7.25 |
| org.slf4j | slf4j-jcl | 1.7.25 |
| org.slf4j  org.slf4j | slf4j-jdk14  slf4j-log4j12 | 1.7.25  1.7.25 |
| org.slf4j  org.slf4j | slf4j-nop  slf4j-simple | 1.7.25  1.7.25 |
| org.spockframework | spock-core | 1.0-groovy-2.4 |
| org.spockframework | spock-spring | 1.0-groovy-2.4 |
| org.springframework | spring-aop | 4.3.12.RELEASE |
| org.springframework  org.springframework | spring-aspects  spring-beans | 4.3.12.RELEASE  4.3.12.RELEASE |
| org.springframework | spring-context | 4.3.12.RELEASE |
| org.springframework | spring-context-support | 4.3.12.RELEASE |
| org.springframework | spring-core | 4.3.12.RELEASE |
| org.springframework  org.springframework | spring-expression  spring-instrument | 4.3.12.RELEASE  4.3.12.RELEASE |
| org.springframework | spring-instrument- tomcat | 4.3.12.RELEASE |
| org.springframework | spring-jdbc | 4.3.12.RELEASE |
| org.springframework  org.springframework | spring-jms  springloaded | 4.3.12.RELEASE  1.2.8.RELEASE |
| org.springframework  org.springframework | spring-messaging  spring-orm | 4.3.12.RELEASE  4.3.12.RELEASE |
| org.springframework | spring-oxm | 4.3.12.RELEASE |
| org.springframework  org.springframework | spring-test  spring-tx | 4.3.12.RELEASE  4.3.12.RELEASE |
| org.springframework | spring-web | 4.3.12.RELEASE |
| org.springframework | spring-webmvc | 4.3.12.RELEASE |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.springframework  org.springframework | spring-webmvc-portlet  spring-websocket | 4.3.12.RELEASE  4.3.12.RELEASE |
| org.springframework.amqp  org.springframework.amqp | spring-amqp  spring-rabbit | 1.7.4.RELEASE  1.7.4.RELEASE |
| org.springframework.batc | hspring-batch-core 3.0.8.RELEASE | |
| org.springframework.batc  org.springframework.batc | hspring-batch- infrastructure  hspring-batch- integration | 3.0.8.RELEASE  3.0.8.RELEASE |
| org.springframework.batc  org.springframework.boot | hspring-batch-test  spring-boot | 3.0.8.RELEASE  1.5.8.RELEASE |
| org.springframework.boot | spring-boot-actuator | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-actuator- docs | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot- autoconfigure | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot- | 1.5.8.RELEASE |
|  | autoconfigure-processor |  |
| org.springframework.boot | spring-boot- | 1.5.8.RELEASE |
|  | configuration-metadata |  |
| org.springframework.boot | spring-boot- configuration-processor | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-devtools | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-loader | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-loader- tools | 1.5.8.RELEASE |
| org.springframework.boot org.springframework.boot | spring-boot-starter  spring-boot-starter- activemq | 1.5.8.RELEASE  1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- actuator | 1.5.8.RELEASE |
| org.springframework.boot  org.springframework.boot | spring-boot-starter- amqp  spring-boot-starter-aop | 1.5.8.RELEASE  1.5.8.RELEASE |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | artemis |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | batch |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | cache |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | cloud-connectors |  |
| org.springframework.boot | spring-boot-starter- data-cassandra | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | data-couchbase |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | data-elasticsearch |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | data-gemfire |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | data-jpa |  |
| org.springframework.boot | spring-boot-starter- data-ldap | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- data-mongodb | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- data-neo4j | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | data-redis |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | data-rest |  |
| org.springframework.boot | spring-boot-starter- data-solr | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- freemarker | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | groovy-templates |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | hateoas |  |

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | integration |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | jdbc |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | jersey |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | jetty |  |
| org.springframework.boot | spring-boot-starter- jooq | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | jta-atomikos |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | jta-bitronix |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | jta-narayana |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | log4j2 |  |
| org.springframework.boot | spring-boot-starter- logging | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- mail | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- mobile | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | mustache |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | remote-shell |  |
| org.springframework.boot | spring-boot-starter- security | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- social-facebook | 1.5.8.RELEASE |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | social-linkedin |  |
| org.springframework.boot | spring-boot-starter- | 1.5.8.RELEASE |
|  | social-twitter |  |

**Group ID Artifact ID Version**

org.springframework.boot spring-boot-starter-

test

org.springframework.boot spring-boot-starter-

thymeleaf

org.springframework.boot spring-boot-starter-

tomcat

org.springframework.boot spring-boot-starter-

undertow

org.springframework.boot spring-boot-starter-

validation

1.5.8.RELEASE

1.5.8.RELEASE

1.5.8.RELEASE

1.5.8.RELEASE

1.5.8.RELEASE

org.springframework.boot spring-boot-starter-web 1.5.8.RELEASE

org.springframework.boot spring-boot-starter-

web-services

org.springframework.boot spring-boot-starter-

websocket

1.5.8.RELEASE

1.5.8.RELEASE

org.springframework.boot spring-boot-test 1.5.8.RELEASE

org.springframework.boot spring-boot-test-

autoconfigure

org.springframework.cloudspring-cloud-

cloudfoundry-connector

1.5.8.RELEASE

1.2.4.RELEASE

org.springframework.cloudspring-cloud-core 1.2.4.RELEASE

org.springframework.cloudspring-cloud-heroku-

connector

org.springframework.cloudspring-cloud-

localconfig-connector

org.springframework.cloudspring-cloud-spring-

service-connector

1.2.4.RELEASE

1.2.4.RELEASE

1.2.4.RELEASE

org.springframework.data spring-cql 1.5.8.RELEASE org.springframework.data spring-data-cassandra 1.5.8.RELEASE org.springframework.data spring-data-commons 1.13.8.RELEASE org.springframework.data spring-data-couchbase 2.2.8.RELEASE

org.springframework.data spring-data-

elasticsearch

2.1.8.RELEASE

org.springframework.data spring-data-envers 1.1.8.RELEASE

org.springframework.data spring-data-gemfire 1.9.8.RELEASE

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.springframework.data | spring-data-jpa | 1.11.8.RELEASE |
| org.springframework.data | spring-data-keyvalue | 1.2.8.RELEASE |
| org.springframework.data | spring-data-ldap | 1.0.8.RELEASE |
| org.springframework.data | spring-data-mongodb | 1.10.8.RELEASE |
| org.springframework.data | spring-data-mongodb- cross-store | 1.10.8.RELEASE |
| org.springframework.data | spring-data-mongodb- log4j | 1.10.8.RELEASE |
| org.springframework.data | spring-data-neo4j | 4.2.8.RELEASE |
| org.springframework.data | spring-data-redis | 1.8.8.RELEASE |
| org.springframework.data | spring-data-rest-core | 2.6.8.RELEASE |
| org.springframework.data | spring-data-rest-hal- browser | 2.6.8.RELEASE |
| org.springframework.data | spring-data-rest-webmvc | 2.6.8.RELEASE |
| org.springframework.data | spring-data-solr | 2.1.8.RELEASE |

org.springframework.hateoasspring-hateoas 0.23.0.RELEASE org.springframework.integrsaptrinng-integration-amqp 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-core 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

event

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-feed 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-file 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-ftp 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

gemfire

org.springframework.integrsaptrinng-integration-

groovy

4.3.12.RELEASE

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-http 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-ip 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

java-dsl

1.2.3.RELEASE

org.springframework.integrsaptrinng-integration-jdbc 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-jms 4.3.12.RELEASE

**Group ID Artifact ID Version** org.springframework.integrsaptrinng-integration-jmx 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-jpa 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-mail 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

mongodb

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-mqtt 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

redis

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-rmi 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

scripting

org.springframework.integrsaptrinng-integration-

security

4.3.12.RELEASE

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-sftp 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

stomp

org.springframework.integrsaptrinng-integration-

stream

org.springframework.integrsaptrinng-integration-

syslog

4.3.12.RELEASE

4.3.12.RELEASE

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-test 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

twitter

org.springframework.integrsaptrinng-integration-

websocket

4.3.12.RELEASE

4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-ws 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-xml 4.3.12.RELEASE org.springframework.integrsaptrinng-integration-xmpp 4.3.12.RELEASE

org.springframework.integrsaptrinng-integration-

zookeeper

4.3.12.RELEASE

org.springframework.kafkaspring-kafka 1.1.7.RELEASE org.springframework.kafkaspring-kafka-test 1.1.7.RELEASE org.springframework.ldap spring-ldap-core 2.3.2.RELEASE org.springframework.ldap spring-ldap-core-tiger 2.3.2.RELEASE

**Group ID Artifact ID Version** org.springframework.ldap spring-ldap-ldif-batch 2.3.2.RELEASE org.springframework.ldap spring-ldap-ldif-core 2.3.2.RELEASE org.springframework.ldap spring-ldap-odm 2.3.2.RELEASE

org.springframework.ldap spring-ldap-test 2.3.2.RELEASE org.springframework.mobilespring-mobile-device 1.1.5.RELEASE org.springframework.pluginspring-plugin-core 1.2.0.RELEASE org.springframework.pluginspring-plugin-metadata 1.2.0.RELEASE org.springframework.restdoscpsring-restdocs-core 1.1.3.RELEASE org.springframework.restdoscpsring-restdocs-mockmvc 1.1.3.RELEASE

org.springframework.restdoscpsring-restdocs-

restassured

1.1.3.RELEASE

org.springframework.retryspring-retry 1.2.1.RELEASE org.springframework.securistpyring-security-acl 4.2.3.RELEASE org.springframework.securistpyring-security-aspects 4.2.3.RELEASE org.springframework.securistpyring-security-cas 4.2.3.RELEASE org.springframework.securistpyring-security-config 4.2.3.RELEASE org.springframework.securistpyring-security-core 4.2.3.RELEASE org.springframework.securistpyring-security-crypto 4.2.3.RELEASE org.springframework.securistpyring-security-data 4.2.3.RELEASE org.springframework.securistpyring-security-jwt 1.0.8.RELEASE org.springframework.securistpyring-security-ldap 4.2.3.RELEASE

org.springframework.securistpyring-security-

messaging

4.2.3.RELEASE

org.springframework.securistpyring-security-openid 4.2.3.RELEASE

org.springframework.securistpyring-security-

remoting

4.2.3.RELEASE

org.springframework.securistpyring-security-taglibs 4.2.3.RELEASE org.springframework.securistpyring-security-test 4.2.3.RELEASE org.springframework.securistpyring-security-web 4.2.3.RELEASE org.springframework.securistpyr.ionagu-tshecurity-oauth 2.0.14.RELEASE org.springframework.securistpyr.ionagu-tshecurity-oauth2 2.0.14.RELEASE

**Group ID Artifact ID Version**

org.springframework.sessiosnpring-session 1.3.1.RELEASE

org.springframework.sessiosnpring-session-data-

gemfire

org.springframework.sessiosnpring-session-data-

mongo

org.springframework.sessiosnpring-session-data-

redis

org.springframework.sessiosnpring-session-

hazelcast

1.3.1.RELEASE

1.3.1.RELEASE

1.3.1.RELEASE

1.3.1.RELEASE

org.springframework.sessiosnpring-session-jdbc 1.3.1.RELEASE org.springframework.socialspring-social-config 1.1.4.RELEASE org.springframework.socialspring-social-core 1.1.4.RELEASE org.springframework.socialspring-social-facebook 2.0.3.RELEASE

org.springframework.socialspring-social-facebook-

web

2.0.3.RELEASE

org.springframework.socialspring-social-linkedin 1.0.2.RELEASE org.springframework.socialspring-social-security 1.1.4.RELEASE org.springframework.socialspring-social-twitter 1.1.2.RELEASE org.springframework.socialspring-social-web 1.1.4.RELEASE org.springframework.ws spring-ws-core 2.4.0.RELEASE org.springframework.ws spring-ws-security 2.4.0.RELEASE org.springframework.ws spring-ws-support 2.4.0.RELEASE org.springframework.ws spring-ws-test 2.4.0.RELEASE org.thymeleaf thymeleaf 2.1.5.RELEASE org.thymeleaf thymeleaf-spring4 2.1.5.RELEASE

org.thymeleaf.extras thymeleaf-extras-

conditionalcomments

org.thymeleaf.extras thymeleaf-extras-

java8time

org.thymeleaf.extras thymeleaf-extras-

springsecurity4

2.1.2.RELEASE

2.1.0.RELEASE

2.1.3.RELEASE

org.webjars hal-browser 9f96c74

org.webjars webjars-locator 0.32-1

|  |  |  |
| --- | --- | --- |
| **Group ID** | **Artifact ID** | **Version** |
| org.xerial | sqlite-jdbc | 3.15.1 |
| org.yaml | snakeyaml | 1.17 |
| redis.clients | jedis | 2.9.0 |
| wsdl4j | wsdl4j | 1.6.3 |
| xml-apis | xml-apis | 1.4.01 |