**Spring Boot Reference Guide**

**Authors**

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

**1.5.7.RELEASE**

Copyright © 2012-2017

*Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically（电子版本的）.*

**Table of Contents**

[**I. Spring Boot Documentation**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation)

[**1. About the documentation**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation-about)

[**2. Getting help**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation-getting-help)

[**3. First steps**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation-first-steps)

[**4. Working with Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_working_with_spring_boot)

[**5. Learning about Spring Boot features**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_learning_about_spring_boot_features)

[**6. Moving to production**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_moving_to_production)

[**7. Advanced topics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_advanced_topics)

[**II. Getting started**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started)

[**8. Introducing Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-introducing-spring-boot)

[**9. System Requirements**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-system-requirements)

[**9.1. Servlet containers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_servlet_containers)

[**10. Installing Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-spring-boot)

[**10.1. Installation instructions for the Java developer**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installation-instructions-for-java)

[10.1.1. Maven installation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-maven-installation)

[10.1.2. Gradle installation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-gradle-installation)

[**10.2. Installing the Spring Boot CLI**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-the-cli)

[10.2.1. Manual installation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-manual-cli-installation)

[10.2.2. Installation with SDKMAN!](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-sdkman-cli-installation)

[10.2.3. OSX Homebrew installation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-homebrew-cli-installation)

[10.2.4. MacPorts installation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-macports-cli-installation)

[10.2.5. Command-line completion](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-cli-command-line-completion)

[10.2.6. Quick start Spring CLI example](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-cli-example)

[**10.3. Upgrading from an earlier version of Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-upgrading-from-an-earlier-version)

[**11. Developing your first Spring Boot application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application)

[**11.1. Creating the POM**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-pom)

[**11.2. Adding classpath dependencies**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-dependencies)

[**11.3. Writing the code**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-code)

[11.3.1. The @RestController and @RequestMapping annotations](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-annotations)

[11.3.2. The @EnableAutoConfiguration annotation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-auto-configuration)

[11.3.3. The “main” method](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-main-method)

[**11.4. Running the example**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-run)

[**11.5. Creating an executable jar**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-executable-jar)

[**12. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-whats-next)

[**III. Using Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot)

[**13. Build systems**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-build-systems)

[**13.1. Dependency management**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-dependency-management)

[**13.2. Maven**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven)

[13.2.1. Inheriting the starter parent](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-parent-pom)

[13.2.2. Using Spring Boot without the parent POM](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-without-a-parent)

[13.2.3. Changing the Java version](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-java-version)

[13.2.4. Using the Spring Boot Maven plugin](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-plugin)

[**13.3. Gradle**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-gradle)

[**13.4. Ant**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-ant)

[**13.5. Starters**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter)

[**14. Structuring your code**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-structuring-your-code)

[**14.1. Using the “default” package**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-using-the-default-package)

[**14.2. Locating the main application class**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-locating-the-main-class)

[**15. Configuration classes**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-configuration-classes)

[**15.1. Importing additional configuration classes**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-importing-configuration)

[**15.2. Importing XML configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-importing-xml-configuration)

[**16. Auto-configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-auto-configuration)

[**16.1. Gradually replacing auto-configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-replacing-auto-configuration)

[**16.2. Disabling specific auto-configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-disabling-specific-auto-configuration)

[**17. Spring Beans and dependency injection**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-spring-beans-and-dependency-injection)

[**18. Using the @SpringBootApplication annotation**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-using-springbootapplication-annotation)

[**19. Running your application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-your-application)

[**19.1. Running from an IDE**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-from-an-ide)

[**19.2. Running as a packaged application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-as-a-packaged-application)

[**19.3. Using the Maven plugin**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-with-the-maven-plugin)

[**19.4. Using the Gradle plugin**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-with-the-gradle-plugin)

[**19.5. Hot swapping**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-hot-swapping)

[**20. Developer tools**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools)

[**20.1. Property defaults**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-property-defaults)

[**20.2. Automatic restart**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart)

[20.2.1. Excluding resources](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-exclude)

[20.2.2. Watching additional paths](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-additional-paths)

[20.2.3. Disabling restart](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-disable)

[20.2.4. Using a trigger file](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-triggerfile)

[20.2.5. Customizing the restart classloader](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-customizing-classload)

[20.2.6. Known limitations](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-known-restart-limitations)

[**20.3. LiveReload**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-livereload)

[**20.4. Global settings**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-globalsettings)

[**20.5. Remote applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-remote)

[20.5.1. Running the remote client application](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_running_the_remote_client_application)

[20.5.2. Remote update](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-remote-update)

[20.5.3. Remote debug tunnel](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-remote-debugtunnel)

[**21. Packaging your application for production**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-packaging-for-production)

[**22. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-whats-next)

[**IV. Spring Boot features**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features)

[**23. SpringApplication**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-application)

[**23.1. Startup failure**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_startup_failure)

[**23.2. Customizing the Banner**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-banner)

[**23.3. Customizing SpringApplication**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-customizing-spring-application)

[**23.4. Fluent builder API**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-fluent-builder-api)

[**23.5. Application events and listeners**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-application-events-and-listeners)

[**23.6. Web environment**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-web-environment)

[**23.7. Accessing application arguments**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-application-arguments)

[**23.8. Using the ApplicationRunner or CommandLineRunner**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-command-line-runner)

[**23.9. Application exit**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-application-exit)

[**23.10. Admin features**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-application-admin)

[**24. Externalized Configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config)

[**24.1. Configuring random values**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-random-values)

[**24.2. Accessing command line properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-command-line-args)

[**24.3. Application property files**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-application-property-files)

[**24.4. Profile-specific properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-profile-specific-properties)

[**24.5. Placeholders in properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-placeholders-in-properties)

[**24.6. Using YAML instead of Properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-yaml)

[24.6.1. Loading YAML](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-loading-yaml)

[24.6.2. Exposing YAML as properties in the Spring Environment](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-exposing-yaml-to-spring)

[24.6.3. Multi-profile YAML documents](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-multi-profile-yaml)

[24.6.4. YAML shortcomings](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-yaml-shortcomings)

[24.6.5. Merging YAML lists](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-complex-type-merge)

[**24.7. Type-safe Configuration Properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-typesafe-configuration-properties)

[24.7.1. Third-party configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-3rd-party-configuration)

[24.7.2. Relaxed binding](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-relaxed-binding)

[24.7.3. Properties conversion](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-conversion)

[24.7.4. @ConfigurationProperties Validation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-validation)

[24.7.5. @ConfigurationProperties vs. @Value](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-vs-value)

[**25. Profiles**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-profiles)

[**25.1. Adding active profiles**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-adding-active-profiles)

[**25.2. Programmatically setting profiles**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-programmatically-setting-profiles)

[**25.3. Profile-specific configuration files**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-profile-specific-configuration)

[**26. Logging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging)

[**26.1. Log format**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging-format)

[**26.2. Console output**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging-console-output)

[26.2.1. Color-coded output](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging-color-coded-output)

[**26.3. File output**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging-file-output)

[**26.4. Log Levels**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-log-levels)

[**26.5. Custom log configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-log-configuration)

[**26.6. Logback extensions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logback-extensions)

[26.6.1. Profile-specific configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_profile_specific_configuration)

[26.6.2. Environment properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_environment_properties)

[**27. Developing web applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-developing-web-applications)

[**27.1. The ‘Spring Web MVC framework’**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc)

[27.1.1. Spring MVC auto-configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-auto-configuration)

[27.1.2. HttpMessageConverters](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-message-converters)

[27.1.3. Custom JSON Serializers and Deserializers](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-json-components)

[27.1.4. MessageCodesResolver](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-message-codes)

[27.1.5. Static Content](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-static-content)

[27.1.6. Custom Favicon](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-favicon)

[27.1.7. ConfigurableWebBindingInitializer](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-web-binding-initializer)

[27.1.8. Template engines](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-template-engines)

[27.1.9. Error Handling](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling)

[Custom error pages](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling-custom-error-pages)

[Mapping error pages outside of Spring MVC](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling-mapping-error-pages-without-mvc)

[Error Handling on WebSphere Application Server](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling-websphere)

[27.1.10. Spring HATEOAS](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-hateoas)

[27.1.11. CORS support](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-cors)

[**27.2. JAX-RS and Jersey**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jersey)

[**27.3. Embedded servlet container support**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container)

[27.3.1. Servlets, Filters, and listeners](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container-servlets-filters-listeners)

[Registering Servlets, Filters, and listeners as Spring beans](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container-servlets-filters-listeners-beans)

[27.3.2. Servlet Context Initialization](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container-context-initializer)

[Scanning for Servlets, Filters, and listeners](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container-servlets-filters-listeners-scanning)

[27.3.3. The EmbeddedWebApplicationContext](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container-application-context)

[27.3.4. Customizing embedded servlet containers](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-customizing-embedded-containers)

[Programmatic customization](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-programmatic-embedded-container-customization)

[Customizing ConfigurableEmbeddedServletContainer directly](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-customizing-configurableembeddedservletcontainerfactory-directly)

[27.3.5. JSP limitations](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jsp-limitations)

[**28. Security**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security)

[**28.1. OAuth2**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-oauth2)

[28.1.1. Authorization Server](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-oauth2-authorization-server)

[28.1.2. Resource Server](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-oauth2-resource-server)

[**28.2. Token Type in User Info**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-oauth2-token-type)

[**28.3. Customizing the User Info RestTemplate**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-custom-user-info)

[28.3.1. Client](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-custom-user-info-client)

[28.3.2. Single Sign On](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-oauth2-single-sign-on)

[**28.4. Actuator Security**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security-actuator)

[**29. Working with SQL databases**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-sql)

[**29.1. Configure a DataSource**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-configure-datasource)

[29.1.1. Embedded Database Support](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-database-support)

[29.1.2. Connection to a production database](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connect-to-production-database)

[29.1.3. Connection to a JNDI DataSource](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-a-jndi-datasource)

[**29.2. Using JdbcTemplate**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-using-jdbc-template)

[**29.3. JPA and ‘Spring Data’**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jpa-and-spring-data)

[29.3.1. Entity Classes](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-entity-classes)

[29.3.2. Spring Data JPA Repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-jpa-repositories)

[29.3.3. Creating and dropping JPA databases](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-creating-and-dropping-jpa-databases)

[29.3.4. Open EntityManager in View](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jpa-in-web-environment)

[**29.4. Using H2’s web console**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-sql-h2-console)

[29.4.1. Changing the H2 console’s path](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-sql-h2-console-custom-path)

[29.4.2. Securing the H2 console](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-sql-h2-console-securing)

[**29.5. Using jOOQ**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jooq)

[29.5.1. Code Generation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_code_generation)

[29.5.2. Using DSLContext](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_using_dslcontext)

[29.5.3. Customizing jOOQ](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_customizing_jooq)

[**30. Working with NoSQL technologies**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-nosql)

[**30.1. Redis**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-redis)

[30.1.1. Connecting to Redis](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-redis)

[**30.2. MongoDB**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-mongodb)

[30.2.1. Connecting to a MongoDB database](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-mongodb)

[30.2.2. MongoTemplate](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-mongo-template)

[30.2.3. Spring Data MongoDB repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-mongo-repositories)

[30.2.4. Embedded Mongo](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-mongo-embedded)

[**30.3. Neo4j**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-neo4j)

[30.3.1. Connecting to a Neo4j database](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-neo4j)

[30.3.2. Using the embedded mode](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-neo4j-embedded)

[30.3.3. Neo4jSession](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-neo4j-ogm-session)

[30.3.4. Spring Data Neo4j repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-neo4j-repositories)

[30.3.5. Repository example](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_repository_example)

[**30.4. Gemfire**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-gemfire)

[**30.5. Solr**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-solr)

[30.5.1. Connecting to Solr](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-solr)

[30.5.2. Spring Data Solr repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-solr-repositories)

[**30.6. Elasticsearch**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-elasticsearch)

[30.6.1. Connecting to Elasticsearch using Jest](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-elasticsearch-jest)

[30.6.2. Connecting to Elasticsearch using Spring Data](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-elasticsearch-spring-data)

[30.6.3. Spring Data Elasticsearch repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-elasticsearch-repositories)

[**30.7. Cassandra**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-cassandra)

[30.7.1. Connecting to Cassandra](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-cassandra)

[30.7.2. Spring Data Cassandra repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-cassandra-repositories)

[**30.8. Couchbase**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-couchbase)

[30.8.1. Connecting to Couchbase](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-couchbase)

[30.8.2. Spring Data Couchbase repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-data-couchbase-repositories)

[**30.9. LDAP**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-ldap)

[30.9.1. Connecting to an LDAP server](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-ldap-connecting)

[30.9.2. Spring Data LDAP repositories](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-ldap-spring-data-repositories)

[30.9.3. Embedded in-memory LDAP server](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-ldap-embedded)

[**31. Caching**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching)

[**31.1. Supported cache providers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_supported_cache_providers)

[31.1.1. Generic](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-generic)

[31.1.2. JCache (JSR-107)](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-jcache)

[31.1.3. EhCache 2.x](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-ehcache2)

[31.1.4. Hazelcast](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-hazelcast)

[31.1.5. Infinispan](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-infinispan)

[31.1.6. Couchbase](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-couchbase)

[31.1.7. Redis](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-redis)

[31.1.8. Caffeine](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-caffeine)

[31.1.9. Guava (deprecated)](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-guava)

[31.1.10. Simple](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-simple)

[31.1.11. None](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-none)

[**32. Messaging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-messaging)

[**32.1. JMS**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jms)

[32.1.1. ActiveMQ support](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-activemq)

[32.1.2. Artemis support](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-artemis)

[32.1.3. Using a JNDI ConnectionFactory](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jms-jndi)

[32.1.4. Sending a message](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-using-jms-sending)

[32.1.5. Receiving a message](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-using-jms-receiving)

[**32.2. AMQP**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-amqp)

[32.2.1. RabbitMQ support](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-rabbitmq)

[32.2.2. Sending a message](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-using-amqp-sending)

[32.2.3. Receiving a message](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-using-amqp-receiving)

[**32.3. Apache Kafka Support**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-kafka)

[32.3.1. Sending a Message](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-kafka-sending-a-message)

[32.3.2. Receiving a Message](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-kafka-receiving-a-message)

[32.3.3. Additional Kafka Properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-kafka-extra-props)

[**33. Calling REST services**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-restclient)

[**33.1. RestTemplate customization**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-restclient-customization)

[**34. Validation**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-validation)

[**35. Sending email**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-email)

[**36. Distributed Transactions with JTA**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta)

[**36.1. Using an Atomikos transaction manager**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta-atomikos)

[**36.2. Using a Bitronix transaction manager**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta-bitronix)

[**36.3. Using a Narayana transaction manager**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta-narayana)

[**36.4. Using a Java EE managed transaction manager**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta-javaee)

[**36.5. Mixing XA and non-XA JMS connections**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta-mixed-jms)

[**36.6. Supporting an alternative embedded transaction manager**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta-supporting-alternative-embedded)

[**37. Hazelcast**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-hazelcast)

[**38. Spring Integration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-integration)

[**39. Spring Session**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-session)

[**40. Monitoring and management over JMX**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jmx)

[**41. Testing**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing)

[**41.1. Test scope dependencies**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-test-scope-dependencies)

[**41.2. Testing Spring applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-applications)

[**41.3. Testing Spring Boot applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications)

[41.3.1. Detecting test configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-detecting-config)

[41.3.2. Excluding test configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-excluding-config)

[41.3.3. Working with random ports](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-working-with-random-ports)

[41.3.4. Mocking and spying beans](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-mocking-beans)

[41.3.5. Auto-configured tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-tests)

[41.3.6. Auto-configured JSON tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-json-tests)

[41.3.7. Auto-configured Spring MVC tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-mvc-tests)

[41.3.8. Auto-configured Data JPA tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-jpa-test)

[41.3.9. Auto-configured JDBC tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-jdbc-test)

[41.3.10. Auto-configured Data MongoDB tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-mongo-test)

[41.3.11. Auto-configured REST clients](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-rest-client)

[41.3.12. Auto-configured Spring REST Docs tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-rest-docs)

[41.3.13. Using Spock to test Spring Boot applications](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-with-spock)

[**41.4. Test utilities**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-test-utilities)

[41.4.1. ConfigFileApplicationContextInitializer](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-configfileapplicationcontextinitializer-test-utility)

[41.4.2. EnvironmentTestUtils](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-environment-test-utilities)

[41.4.3. OutputCapture](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-output-capture-test-utility)

[41.4.4. TestRestTemplate](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-rest-templates-test-utility)

[**42. WebSockets**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-websockets)

[**43. Web Services**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-webservices)

[**44. Creating your own auto-configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-developing-auto-configuration)

[**44.1. Understanding auto-configured beans**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-understanding-auto-configured-beans)

[**44.2. Locating auto-configuration candidates**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-locating-auto-configuration-candidates)

[**44.3. Condition annotations**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-condition-annotations)

[44.3.1. Class conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-class-conditions)

[44.3.2. Bean conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-bean-conditions)

[44.3.3. Property conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-property-conditions)

[44.3.4. Resource conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-resource-conditions)

[44.3.5. Web application conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-web-application-conditions)

[44.3.6. SpEL expression conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spel-conditions)

[**44.4. Creating your own starter**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter)

[44.4.1. Naming](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter-naming)

[44.4.2. Autoconfigure module](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter-module-autoconfigure)

[44.4.3. Starter module](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter-module-starter)

[**45. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-whats-next)

[**V. Spring Boot Actuator: Production-ready features**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready)

[**46. Enabling production-ready features**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-enabling)

[**47. Endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints)

[**47.1. Customizing endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-endpoints)

[**47.2. Hypermedia for actuator MVC endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoint-hypermedia)

[**47.3. CORS support**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoint-cors)

[**47.4. Adding custom endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-endpoints-programmatically)

[**47.5. Health information**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-health)

[**47.6. Security with HealthIndicators**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_security_with_healthindicators)

[47.6.1. Auto-configured HealthIndicators](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_auto_configured_healthindicators)

[47.6.2. Writing custom HealthIndicators](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_writing_custom_healthindicators)

[**47.7. Application information**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info)

[47.7.1. Auto-configured InfoContributors](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info-autoconfigure)

[47.7.2. Custom application info information](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info-env)

[47.7.3. Git commit information](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info-git)

[47.7.4. Build information](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info-build)

[47.7.5. Writing custom InfoContributors](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info-custom)

[**48. Monitoring and management over HTTP**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-monitoring)

[**48.1. Accessing sensitive endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-sensitive-endpoints)

[**48.2. Customizing the management endpoint paths**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-management-server-context-path)

[**48.3. Customizing the management server port**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-management-server-port)

[**48.4. Configuring management-specific SSL**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-management-specific-ssl)

[**48.5. Customizing the management server address**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-management-server-address)

[**48.6. Disabling HTTP endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-disabling-http-endpoints)

[**48.7. HTTP health endpoint format and access restrictions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-health-access-restrictions)

[**49. Monitoring and management over JMX**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-jmx)

[**49.1. Customizing MBean names**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-custom-mbean-names)

[**49.2. Disabling JMX endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-disable-jmx-endpoints)

[**49.3. Using Jolokia for JMX over HTTP**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-jolokia)

[49.3.1. Customizing Jolokia](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-jolokia)

[49.3.2. Disabling Jolokia](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-disabling-jolokia)

[**50. Monitoring and management using a remote shell (deprecated)**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-remote-shell)

[**50.1. Connecting to the remote shell**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-connecting-to-the-remote-shell)

[50.1.1. Remote shell credentials](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-remote-shell-credentials)

[**50.2. Extending the remote shell**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-extending-the-remote-shell)

[50.2.1. Remote shell commands](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-remote-commands)

[50.2.2. Remote shell plugins](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-remote-shell-plugins)

[**51. Loggers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-loggers)

[**51.1. Configure a Logger**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-logger-configuration)

[**52. Metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metrics)

[**52.1. System metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-system-metrics)

[**52.2. DataSource metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-datasource-metrics)

[**52.3. Cache metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-datasource-cache)

[**52.4. Tomcat session metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-session-metrics)

[**52.5. Recording your own metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-recording-metrics)

[**52.6. Adding your own public metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-public-metrics)

[**52.7. Special features with Java 8**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-repositories)

[**52.8. Metric writers, exporters and aggregation**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-writers)

[52.8.1. Example: Export to Redis](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-writers-export-to-redis)

[52.8.2. Example: Export to Open TSDB](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-writers-export-to-open-tsdb)

[52.8.3. Example: Export to Statsd](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-writers-export-to-statsd)

[52.8.4. Example: Export to JMX](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-writers-export-to-jmx)

[**52.9. Aggregating metrics from multiple sources**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metric-aggregation)

[**52.10. Dropwizard Metrics**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-dropwizard-metrics)

[**52.11. Message channel integration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metrics-message-channel-integration)

[**53. Auditing**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-auditing)

[**54. Tracing**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-tracing)

[**54.1. Custom tracing**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-custom-tracing)

[**55. Process monitoring**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-process-monitoring)

[**55.1. Extend configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-process-monitoring-configuration)

[**55.2. Programmatically**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-process-monitoring-programmatically)

[**56. Cloud Foundry support**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-cloudfoundry)

[**56.1. Disabling extended Cloud Foundry actuator support**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-cloudfoundry-disable)

[**56.2. Cloud Foundry self signed certificates**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-cloudfoundry-ssl)

[**56.3. Custom security configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-cloudfoundry-custom-security)

[**57. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-whats-next)

[**VI. Deploying Spring Boot applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment)

[**58. Deploying to the cloud**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment)

[**58.1. Cloud Foundry**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-cloud-foundry)

[58.1.1. Binding to services](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-cloud-foundry-services)

[**58.2. Heroku**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-heroku)

[**58.3. OpenShift**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-openshift)

[**58.4. Amazon Web Services (AWS)**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-aws)

[58.4.1. AWS Elastic Beanstalk](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_aws_elastic_beanstalk)

[Using the Tomcat platform](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_using_the_tomcat_platform)

[Using the Java SE platform](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_using_the_java_se_platform)

[Best practices](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_best_practices)

[58.4.2. Summary](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_summary)

[**58.5. Boxfuse and Amazon Web Services**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-boxfuse)

[**58.6. Google Cloud**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment-gae)

[**59. Installing Spring Boot applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-install)

[**59.1. Supported operating systems**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-install-supported-operating-systems)

[**59.2. Unix/Linux services**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-service)

[59.2.1. Installation as an init.d service (System V)](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-initd-service)

[Securing an init.d service](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-initd-service-securing)

[59.2.2. Installation as a systemd service](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-systemd-service)

[59.2.3. Customizing the startup script](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-script-customization)

[Customizing script when it’s written](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-script-customization-when-it-written)

[Customizing script when it runs](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-script-customization-when-it-runs)

[**59.3. Microsoft Windows services**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-windows)

[**60. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-whats-next)

[**VII. Spring Boot CLI**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli)

[**61. Installing the CLI**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-installation)

[**62. Using the CLI**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-using-the-cli)

[**62.1. Running applications using the CLI**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-run)

[62.1.1. Deduced “grab” dependencies](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-deduced-grab-annotations)

[62.1.2. Deduced “grab” coordinates](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-default-grab-deduced-coordinates)

[62.1.3. Default import statements](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-default-import-statements)

[62.1.4. Automatic main method](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-automatic-main-method)

[62.1.5. Custom dependency management](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-default-grab-deduced-coordinates-custom-dependency-management)

[**62.2. Testing your code**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-testing)

[**62.3. Applications with multiple source files**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-multiple-source-files)

[**62.4. Packaging your application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-jar)

[**62.5. Initialize a new project**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-init)

[**62.6. Using the embedded shell**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-shell)

[**62.7. Adding extensions to the CLI**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-install-uninstall)

[**63. Developing application with the Groovy beans DSL**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-groovy-beans-dsl)

[**64. Configuring the CLI with settings.xml**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-maven-settings)

[**65. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli-whats-next)

[**VIII. Build tool plugins**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins)

[**66. Spring Boot Maven plugin**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-maven-plugin)

[**66.1. Including the plugin**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-include-maven-plugin)

[**66.2. Packaging executable jar and war files**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-maven-packaging)

[**67. Spring Boot Gradle plugin**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-plugin)

[**67.1. Including the plugin**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-including-the-gradle-plugin)

[**67.2. Gradle dependency management**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-dependency-management)

[**67.3. Packaging executable jar and war files**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-packaging)

[**67.4. Running a project in-place**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-running-applications)

[**67.5. Spring Boot plugin configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-global-configuration)

[**67.6. Repackage configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-repackage-configuration)

[**67.7. Repackage with custom Gradle configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-repackage-custom-configuration)

[67.7.1. Configuration options](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-configuration-options)

[67.7.2. Available layouts](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-configuration-layouts)

[67.7.3. Using a custom layout](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-configuration-custom-repackager)

[**67.8. Understanding how the Gradle plugin works**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-understanding-the-gradle-plugin)

[**67.9. Publishing artifacts to a Maven repository using Gradle**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-publishing-artifacts-to-a-maven-repository)

[67.9.1. Configuring Gradle to produce a pom that inherits dependency management](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-publishing-artifacts-to-a-maven-repository-inherit)

[67.9.2. Configuring Gradle to produce a pom that imports dependency management](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-publishing-artifacts-to-a-maven-repository-import)

[**68. Spring Boot AntLib module**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-antlib)

[**68.1. Spring Boot Ant tasks**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_spring_boot_ant_tasks)

[68.1.1. spring-boot:exejar](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_spring_boot_exejar)

[68.1.2. Examples](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_examples)

[**68.2. spring-boot:findmainclass**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_spring_boot_findmainclass)

[68.2.1. Examples](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_examples_2)

[**69. Supporting other build systems**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-other-build-systems)

[**69.1. Repackaging archives**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-repackaging-archives)

[**69.2. Nested libraries**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-nested-libraries)

[**69.3. Finding a main class**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-find-a-main-class)

[**69.4. Example repackage implementation**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-repackage-implementation)

[**70. What to read next**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-whats-next)

[**IX. ‘How-to’ guides**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto)

[**71. Spring Boot application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-spring-boot-application)

[**71.1. Create your own FailureAnalyzer**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-failure-analyzer)

[**71.2. Troubleshoot auto-configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-troubleshoot-auto-configuration)

[**71.3. Customize the Environment or ApplicationContext before it starts**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-the-environment-or-application-context)

[**71.4. Build an ApplicationContext hierarchy (adding a parent or root context)**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build-an-application-context-hierarchy)

[**71.5. Create a non-web application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-a-non-web-application)

[**72. Properties & configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-properties-and-configuration)

[**72.1. Automatically expand properties at build time**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-automatic-expansion)

[72.1.1. Automatic property expansion using Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-automatic-expansion-maven)

[72.1.2. Automatic property expansion using Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-automatic-expansion-gradle)

[**72.2. Externalize the configuration of SpringApplication**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-externalize-configuration)

[**72.3. Change the location of external properties of an application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-change-the-location-of-external-properties)

[**72.4. Use ‘short’ command line arguments**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-short-command-line-arguments)

[**72.5. Use YAML for external properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-yaml-for-external-properties)

[**72.6. Set the active Spring profiles**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-set-active-spring-profiles)

[**72.7. Change configuration depending on the environment**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-change-configuration-depending-on-the-environment)

[**72.8. Discover built-in options for external properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-discover-build-in-options-for-external-properties)

[**73. Embedded servlet containers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-embedded-servlet-containers)

[**73.1. Add a Servlet, Filter or Listener to an application**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-add-a-servlet-filter-or-listener)

[73.1.1. Add a Servlet, Filter or Listener using a Spring bean](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-add-a-servlet-filter-or-listener-as-spring-bean)

[Disable registration of a Servlet or Filter](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-disable-registration-of-a-servlet-or-filter)

[73.1.2. Add Servlets, Filters, and Listeners using classpath scanning](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-add-a-servlet-filter-or-listener-using-scanning)

[**73.2. Change the HTTP port**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-change-the-http-port)

[**73.3. Use a random unassigned HTTP port**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-user-a-random-unassigned-http-port)

[**73.4. Discover the HTTP port at runtime**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-discover-the-http-port-at-runtime)

[**73.5. Configure SSL**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-ssl)

[**73.6. Configure Access Logging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-accesslogs)

[**73.7. Use behind a front-end proxy server**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-tomcat-behind-a-proxy-server)

[73.7.1. Customize Tomcat’s proxy configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-tomcat-behind-a-proxy-server)

[**73.8. Configure Tomcat**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-tomcat)

[**73.9. Enable Multiple Connectors with Tomcat**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-enable-multiple-connectors-in-tomcat)

[**73.10. Use Tomcat’s LegacyCookieProcessor**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-tomcat-legacycookieprocessor)

[**73.11. Use Jetty instead of Tomcat**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-instead-of-tomcat)

[**73.12. Configure Jetty**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-jetty)

[**73.13. Use Undertow instead of Tomcat**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-undertow-instead-of-tomcat)

[**73.14. Configure Undertow**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-undertow)

[**73.15. Enable Multiple Listeners with Undertow**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-enable-multiple-listeners-in-undertow)

[**73.16. Use Tomcat 7.x or 8.0**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-tomcat-7)

[73.16.1. Use Tomcat 7.x or 8.0 with Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-tomcat-7-maven)

[73.16.2. Use Tomcat 7.x or 8.0 with Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_use_tomcat_7_x_or_8_0_with_gradle)

[**73.17. Use Jetty 9.2**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-9.2)

[73.17.1. Use Jetty 9.2 with Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-9.2-maven)

[73.17.2. Use Jetty 9.2 with Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-9.2-gradle)

[**73.18. Use Jetty 8**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-8)

[73.18.1. Use Jetty 8 with Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-8-maven)

[73.18.2. Use Jetty 8 with Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-8-gradle)

[**73.19. Create WebSocket endpoints using @ServerEndpoint**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-websocket-endpoints-using-serverendpoint)

[**73.20. Enable HTTP response compression**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#how-to-enable-http-response-compression)

[**74. Spring MVC**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-spring-mvc)

[**74.1. Write a JSON REST service**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-write-a-json-rest-service)

[**74.2. Write an XML REST service**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-write-an-xml-rest-service)

[**74.3. Customize the Jackson ObjectMapper**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-the-jackson-objectmapper)

[**74.4. Customize the @ResponseBody rendering**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-the-responsebody-rendering)

[**74.5. Handling Multipart File Uploads**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-multipart-file-upload-configuration)

[**74.6. Switch off the Spring MVC DispatcherServlet**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-switch-off-the-spring-mvc-dispatcherservlet)

[**74.7. Switch off the Default MVC configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-switch-off-default-mvc-configuration)

[**74.8. Customize ViewResolvers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-view-resolvers)

[**74.9. Use Thymeleaf 3**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-thymeleaf-3)

[**75. HTTP clients**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-http-clients)

[**75.1. Configure RestTemplate to use a proxy**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-http-clients-proxy-configuration)

[**76. Logging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-logging)

[**76.1. Configure Logback for logging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-logback-for-logging)

[76.1.1. Configure logback for file only output](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-logback-for-logging-fileonly)

[**76.2. Configure Log4j for logging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-log4j-for-logging)

[76.2.1. Use YAML or JSON to configure Log4j 2](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-log4j-for-logging-yaml-or-json-config)

[**77. Data Access**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-data-access)

[**77.1. Configure a custom DataSource**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-a-datasource)

[**77.2. Configure Two DataSources**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-two-datasources)

[**77.3. Use Spring Data repositories**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-spring-data-repositories)

[**77.4. Separate @Entity definitions from Spring configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-separate-entity-definitions-from-spring-configuration)

[**77.5. Configure JPA properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-jpa-properties)

[**77.6. Configure Hibernate Naming Strategy**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-hibernate-naming-strategy)

[**77.7. Use a custom EntityManagerFactory**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-custom-entity-manager)

[**77.8. Use Two EntityManagers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-two-entity-managers)

[**77.9. Use a traditional persistence.xml**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-traditional-persistence-xml)

[**77.10. Use Spring Data JPA and Mongo repositories**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-spring-data-jpa--and-mongo-repositories)

[**77.11. Expose Spring Data repositories as REST endpoint**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-exposing-spring-data-repositories-rest-endpoint)

[**77.12. Configure a component that is used by JPA**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-a-component-that-is-used-by-JPA)

[**78. Database initialization**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-database-initialization)

[**78.1. Initialize a database using JPA**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-initialize-a-database-using-jpa)

[**78.2. Initialize a database using Hibernate**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-initialize-a-database-using-hibernate)

[**78.3. Initialize a database using Spring JDBC**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-initialize-a-database-using-spring-jdbc)

[**78.4. Initialize a Spring Batch database**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-initialize-a-spring-batch-database)

[**78.5. Use a higher-level database migration tool**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-a-higher-level-database-migration-tool)

[78.5.1. Execute Flyway database migrations on startup](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-execute-flyway-database-migrations-on-startup)

[78.5.2. Execute Liquibase database migrations on startup](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-execute-liquibase-database-migrations-on-startup)

[**79. Messaging**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-messaging)

[**79.1. Disable transacted JMS session**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-jms-disable-transaction)

[**80. Batch applications**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-batch-applications)

[**80.1. Execute Spring Batch jobs on startup**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-execute-spring-batch-jobs-on-startup)

[**81. Actuator**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-actuator)

[**81.1. Change the HTTP port or address of the actuator endpoints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-change-the-http-port-or-address-of-the-actuator-endpoints)

[**81.2. Customize the ‘whitelabel’ error page**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-the-whitelabel-error-page)

[**81.3. Actuator and Jersey**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-actuator-with-jersey)

[**82. Security**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-security)

[**82.1. Switch off the Spring Boot security configuration**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-switch-off-spring-boot-security-configuration)

[**82.2. Change the AuthenticationManager and add user accounts**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-change-the-authenticationmanager-and-add-user-accounts)

[**82.3. Enable HTTPS when running behind a proxy server**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-enable-https)

[**83. Hot swapping**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-hotswapping)

[**83.1. Reload static content**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-static-content)

[**83.2. Reload templates without restarting the container**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-thymeleaf-template-content)

[83.2.1. Thymeleaf templates](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-thymeleaf-content)

[83.2.2. FreeMarker templates](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-freemarker-content)

[83.2.3. Groovy templates](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-groovy-template-content)

[**83.3. Fast application restarts**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-fast-restart)

[**83.4. Reload Java classes without restarting the container**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-java-classes-without-restarting)

[83.4.1. Configuring Spring Loaded for use with Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-springloaded-maven)

[83.4.2. Configuring Spring Loaded for use with Gradle and IntelliJ IDEA](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-reload-springloaded-gradle-and-intellij-idea)

[**84. Build**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build)

[**84.1. Generate build information**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build-info)

[**84.2. Generate git information**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-git-info)

[**84.3. Customize dependency versions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-dependency-versions)

[**84.4. Create an executable JAR with Maven**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-an-executable-jar-with-maven)

[**84.5. Use a Spring Boot application as a dependency**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-an-additional-executable-jar)

[**84.6. Extract specific libraries when an executable jar runs**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-extract-specific-libraries-when-an-executable-jar-runs)

[**84.7. Create a non-executable JAR with exclusions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-a-nonexecutable-jar)

[**84.8. Remote debug a Spring Boot application started with Maven**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-remote-debug-maven-run)

[**84.9. Remote debug a Spring Boot application started with Gradle**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-remote-debug-gradle-run)

[**84.10. Build an executable archive from Ant without using spring-boot-antlib**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build-an-executable-archive-with-ant)

[**84.11. How to use Java 6**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-java-6)

[84.11.1. Embedded servlet container compatibility](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-java-6-embedded-container)

[84.11.2. Jackson](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-java-6-jackson)

[84.11.3. JTA API compatibility](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#how-to-use-java-6-jta-api)

[**85. Traditional deployment**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-traditional-deployment)

[**85.1. Create a deployable war file**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-a-deployable-war-file)

[**85.2. Create a deployable war file for older servlet containers**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-a-deployable-war-file-for-older-containers)

[**85.3. Convert an existing application to Spring Boot**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-convert-an-existing-application-to-spring-boot)

[**85.4. Deploying a WAR to WebLogic**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-weblogic)

[**85.5. Deploying a WAR in an Old (Servlet 2.5) Container**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-servlet-2-5)

[**X. Appendices**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix)

[**A. Common application properties**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties)

[**B. Configuration meta-data**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata)

[**B.1. Meta-data format**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-format)

[B.1.1. Group Attributes](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-group-attributes)

[B.1.2. Property Attributes](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-property-attributes)

[B.1.3. Hint Attributes](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-hints-attributes)

[B.1.4. Repeated meta-data items](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-repeated-items)

[**B.2. Providing manual hints**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-providing-manual-hints)

[B.2.1. Value hint](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_value_hint)

[B.2.2. Value provider](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_value_provider)

[Any](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_any)

[Class reference](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_class_reference)

[Handle As](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_handle_as)

[Logger name](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_logger_name)

[Spring bean reference](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_spring_bean_reference)

[Spring profile name](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#_spring_profile_name)

[**B.3. Generating your own meta-data using the annotation processor**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-annotation-processor)

[B.3.1. Nested properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-nested-properties)

[B.3.2. Adding additional meta-data](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-additional-metadata)

[**C. Auto-configuration classes**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#auto-configuration-classes)

[**C.1. From the “spring-boot-autoconfigure” module**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#auto-configuration-classes-from-autoconfigure-module)

[**C.2. From the “spring-boot-actuator” module**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#auto-configuration-classes-from-actuator)

[**D. Test auto-configuration annotations**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration)

[**E. The executable jar format**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar)

[**E.1. Nested JARs**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-nested-jars)

[E.1.1. The executable jar file structure](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-jar-file-structure)

[E.1.2. The executable war file structure](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-war-file-structure)

[**E.2. Spring Boot’s “JarFile” class**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-jarfile)

[E.2.1. Compatibility with the standard Java “JarFile”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-jarfile-compatibility)

[**E.3. Launching executable jars**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-launching)

[E.3.1. Launcher manifest](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-launcher-manifest)

[E.3.2. Exploded archives](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-exploded-archives)

[**E.4. PropertiesLauncher Features**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-property-launcher-features)

[**E.5. Executable jar restrictions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-restrictions)

[E.5.1. Zip entry compression](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-zip-entry-compression)

[E.5.2. System ClassLoader](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-system-classloader)

[**E.6. Alternative single jar solutions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-alternatives)

[**F. Dependency versions**](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix-dependency-versions)

**Part I. Spring Boot Documentation**

This section provides a brief overview of Spring Boot reference documentation. Think of it as map for the rest of the document. You can read this reference guide in a linear（线性的方式） fashion, or you can skip sections if something doesn’t interest you.

**1. About the documentation**

The Spring Boot reference guide is available as [html](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/reference/html), [pdf](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/reference/pdf/spring-boot-reference.pdf) and [epub](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/reference/epub/spring-boot-reference.epub) documents. The latest copy is available at [docs.spring.io/spring-boot/docs/current/reference](http://docs.spring.io/spring-boot/docs/current/reference).

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

**2. Getting help**

Having trouble with Spring Boot, We’d like to help!

* Try the [How-to’s](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto) — they provide solutions to the most common questions.
* Learn the Spring basics — Spring Boot builds on many other Spring projects, check the [spring.io](https://spring.io/) web-site for a wealth of reference documentation. If you are just starting out with Spring, try one of the [guides](https://spring.io/guides).
* Ask a question - we monitor [stackoverflow.com](https://stackoverflow.com/) for questions tagged with [spring-boot](https://stackoverflow.com/tags/spring-boot).
* Report bugs with Spring Boot at [github.com/spring-projects/spring-boot/issues](https://github.com/spring-projects/spring-boot/issues).

|  |
| --- |
| [Note] |
| All of Spring Boot is open source, including the documentation! If you find problems with the docs; or if you just want to improve them, please [get involved](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE). |

**3. First steps**

If you’re just getting started with Spring Boot, or 'Spring' in general, [this is the place to start!](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started)

* **From scratch:** [Overview](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-introducing-spring-boot) | [Requirements](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-system-requirements) | [Installation](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-spring-boot)
* **Tutorial:** [Part 1](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application) | [Part 2](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-code)
* **Running your example:** [Part 1](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-run) | [Part 2](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-executable-jar)

**4. Working with Spring Boot**

Ready to actually start using Spring Boot? [We’ve got you covered](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot).

* **Build systems:** [Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven) | [Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-gradle) | [Ant](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-ant) | [Starters](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter)
* **Best practices:** [Code Structure](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-structuring-your-code) | [@Configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-configuration-classes) | [@EnableAutoConfiguration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-auto-configuration) | [Beans and Dependency Injection](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-spring-beans-and-dependency-injection)
* **Running your code** [IDE](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-from-an-ide) | [Packaged](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-as-a-packaged-application) | [Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-with-the-maven-plugin) | [Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-running-with-the-gradle-plugin)
* **Packaging your app:** [Production jars](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-packaging-for-production)
* **Spring Boot CLI:** [Using the CLI](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cli)

**5. Learning about Spring Boot features**

Need more details about Spring Boot’s core features? [This is for you](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features)!

* **Core Features:** [SpringApplication](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-application) | [External Configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config) | [Profiles](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-profiles) | [Logging](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging)
* **Web Applications:** [MVC](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc) | [Embedded Containers](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-embedded-container)
* **Working with data:** [SQL](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-sql) | [NO-SQL](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-nosql)
* **Messaging:** [Overview](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-messaging) | [JMS](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jms)
* **Testing:** [Overview](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing) | [Boot Applications](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications) | [Utils](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-test-utilities)
* **Extending:** [Auto-configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-developing-auto-configuration) | [@Conditions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-condition-annotations)

**6. Moving to production**

When you’re ready to push your Spring Boot application to production, we’ve got [some tricks that you might like](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready)!

* **Management endpoints:** [Overview](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints) | [Customization](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-endpoints)
* **Connection options:** [HTTP](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-monitoring) | [JMX](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-jmx) | [SSH](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-remote-shell)
* **Monitoring:** [Metrics](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metrics) | [Auditing](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-auditing) | [Tracing](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-tracing) | [Process](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-process-monitoring)

**7. Advanced topics**

Lastly, we have a few topics for the more advanced user.

* **Deploy Spring Boot Applications:** [Cloud Deployment](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#cloud-deployment) | [OS Service](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-service)
* **Build tool plugins:** [Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-maven-plugin) | [Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-plugin)
* **Appendix:** [Application Properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties) | [Auto-configuration classes](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#auto-configuration-classes) | [Executable Jars](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar)

**Part II. Getting started**

If you’re just getting started with Spring Boot, or 'Spring' in general, this is the section for you! Here we answer the basic “what?”, “how?” and “why?” questions. You’ll find a gentle introduction to Spring Boot along with installation instructions. We’ll then build our first Spring Boot application, discussing some core principles as we go.

**8. Introducing Spring Boot**

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

You can use Spring Boot to create Java applications that can be started using java -jar or more traditional war deployments. We also provide a command line tool that runs “spring scripts”（提供了运行spring脚本的命令行工具）.

Our primary goals are:

* Provide a radically（彻底的） faster and widely accessible getting started experience for all Spring development.
* Be opinionated out of the box, but get out of the way quickly as requirements start to diverge（分开，分叉） from the defaults.
* Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers（嵌入式服务）, security, metrics（图表）, health checks, externalized configuration).
* Absolutely no code generation and no requirement for XML configuration（完全的无代码差生，无需配置xml）.

**9. System Requirements**

By default, Spring Boot 1.5.7.RELEASE requires [Java 7](http://www.java.com/) and Spring Framework 4.3.11.RELEASE or above. You can use Spring Boot with Java 6 with some additional configuration. See [Section 84.11, “How to use Java 6”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-java-6) for more details. Explicit build support is provided for Maven (3.2+), and Gradle 2 (2.9 or later) and 3.

构建工具需要：maven 3.2+;gradle 2.9+

|  |
| --- |
| [Tip] |
| Although you can use Spring Boot with Java 6 or 7, we generally recommend Java 8 if at all possible. |

**9.1 Servlet containers**

The following embedded servlet containers（嵌入是servlet容器） are supported out of the box(非常好):

| **Name** | **Servlet Version** | **Java Version** |
| --- | --- | --- |
| Tomcat 8 | 3.1 | Java 7+ |
| Tomcat 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+ |

You can also deploy Spring Boot applications to any Servlet 3.0+ compatible container.

可以将spring boot应用部署到任何的兼容Servlet容器中，如tomcat、resin、jetty等。

**10. Installing Spring Boot**

Spring Boot can be used with “classic” Java development tools or installed as a command line tool. Regardless, you will need [Java SDK v1.6](http://www.java.com/) or higher. You should check your current Java installation before you begin:

$ java -version

If you are new to Java development, or if you just want to experiment with Spring Boot you might want to try the [Spring Boot CLI](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-the-cli) first, otherwise, read on for “classic” installation instructions.

|  |
| --- |
| [Tip] |
| Although Spring Boot is compatible with Java 1.6, if possible, you should consider using the latest version of Java. |

**10.1 Installation instructions for the Java developer**

You can use Spring Boot in the same way as any standard Java library. Simply include the appropriate spring-boot-\*.jar files on your classpath. Spring Boot does not require any special tools integration, so you can use any IDE or text editor; and there is nothing special about a Spring Boot application, so you can run and debug as you would any other Java program.

Although you *could* just copy Spring Boot jars, we generally recommend that you use a build tool that supports dependency management (such as Maven or Gradle)（最好使用支持依赖管理的构建工具，如Maven或者Gradle）.

**10.1.1 Maven installation**

Spring Boot is compatible with Apache Maven 3.2 or above. If you don’t already have Maven installed you can follow the instructions at [maven.apache.org](https://maven.apache.org/).

|  |
| --- |
| [Tip] |
| On many operating systems Maven can be installed via a package manager. If you’re an OSX Homebrew user try brew install maven. Ubuntu users can run sudo apt-get install maven. |

Spring Boot dependencies use the org.springframework.boot groupId. Typically your Maven POM file will inherit from the spring-boot-starter-parent project and declare dependencies to one or more [“Starters”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter). Spring Boot also provides an optional [Maven plugin](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-maven-plugin) to create executable jars.

Here is a typical pom.xml file:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 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>1.5.7.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] |
| The spring-boot-starter-parent is a great way to use Spring Boot（主要是为了使用parent项目的配置才继承的相当于gradle中的allprojects）, but it might not be suitable all of the time. Sometimes you may need to inherit from a different parent POM, or you might just not like our default settings. See [Section 13.2.2, “Using Spring Boot without the parent POM”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-without-a-parent) for an alternative solution that uses an import scope. |

**10.1.2 Gradle installation**

Spring Boot is compatible with Gradle 2 (2.9 or later) and Gradle 3. If you don’t already have Gradle installed you can follow the instructions at [www.gradle.org/](http://www.gradle.org/).

Spring Boot dependencies can be declared using the org.springframework.boot group. Typically your project will declare dependencies to one or more [“Starters”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter). Spring Boot provides a useful [Gradle plugin](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-plugin) that can be used to simplify dependency declarations and to create executable jars.

**Gradle Wrapper**

The Gradle Wrapper provides a nice way of “obtaining” Gradle when you need to build a project. It’s a small script and library that you commit alongside your code to bootstrap the build process. See [docs.gradle.org/2.14.1/userguide/gradle\_wrapper.html](https://docs.gradle.org/2.14.1/userguide/gradle_wrapper.html) for details.

Here is a typical build.gradle file:

plugins {

id 'org.springframework.boot' version '1.5.7.RELEASE'

id 'java'

}

jar {

baseName = 'myproject'

version = '0.0.1-SNAPSHOT'

}

repositories {

jcenter()

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

**10.2 Installing the Spring Boot CLI**

The Spring Boot CLI is a command line tool that can be used if you want to quickly prototype(原型) with Spring. It allows you to run [Groovy](http://groovy.codehaus.org/) scripts（可以执行Groovy脚本）, which means that you have a familiar Java-like syntax（类java的语法，Groovy完全兼容java语法）, without so much boilerplate(样本是的，指的是java的强类型) code.

You don’t need to use the CLI to work with Spring Boot but it’s definitely the quickest way to get a Spring application off the ground.

**10.2.1 Manual installation**

You can download the Spring CLI distribution from the Spring software repository:

* [spring-boot-cli-1.5.7.RELEASE-bin.zip](http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.5.7.RELEASE/spring-boot-cli-1.5.7.RELEASE-bin.zip)
* [spring-boot-cli-1.5.7.RELEASE-bin.tar.gz](http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.5.7.RELEASE/spring-boot-cli-1.5.7.RELEASE-bin.tar.gz)

Cutting edge [snapshot distributions](http://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/) are also available.

Once downloaded, follow the [INSTALL.txt](https://raw.github.com/spring-projects/spring-boot/v1.5.7.RELEASE/spring-boot-cli/src/main/content/INSTALL.txt) instructions（遵照install.txt的指示安装） from the unpacked archive. In summary: there is a spring script (spring.bat for Windows) in a bin/ directory in the .zip file, or alternatively you can use java -jar with the .jar file (the script helps you to be sure that the classpath is set correctly).

**10.2.2 Installation with SDKMAN!**

SDKMAN! (The Software Development Kit Manager-软件开发包管理，可以用来管理不同版本的二进制sdk) can be used for managing multiple versions of various binary SDKs, including Groovy and the Spring Boot CLI. Get SDKMAN! from [sdkman.io](http://sdkman.io/) and install Spring Boot with

$ sdk install springboot

$ spring --version

Spring Boot v1.5.7.RELEASE

If you are developing features for the CLI and want easy access to the version you just built, follow these extra instructions.

$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-1.5.7.RELEASE-bin/spring-1.5.7.RELEASE/

$ sdk default springboot dev

$ spring --version

Spring CLI v1.5.7.RELEASE

This will install a local instance of spring called the dev instance. It points at your target build location, so every time you rebuild Spring Boot, spring will be up-to-date.

You can see it by doing this:

$ sdk ls springboot

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

Available Springboot Versions

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

> + dev

\* 1.5.7.RELEASE

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

+ - local version

\* - installed

> - currently in use

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

**10.2.3 OSX Homebrew installation**

If you are on a Mac and using [Homebrew](http://brew.sh/), all you need to do to install the Spring Boot CLI is:

$ brew tap pivotal/tap

$ brew install springboot

Homebrew will install spring to /usr/local/bin.

|  |
| --- |
| [Note] |
| If you don’t see the formula, your installation of brew might be out-of-date. Just execute brew update and try again. |

**10.2.4 MacPorts installation**

If you are on a Mac and using [MacPorts](https://www.macports.org/), all you need to do to install the Spring Boot CLI is:

$ sudo port install spring-boot-cli

**10.2.5 Command-line completion**

代码自动完成/补全配置，也可以在install.txt中找到相关的信息

Spring Boot CLI ships with scripts that provide command completion for [BASH](https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29) and [zsh](https://en.wikipedia.org/wiki/Zsh) shells. You can source the script (also named spring) in any shell, or put it in your personal or system-wide bash completion initialization. On a Debian system the system-wide scripts are in /shell-completion/bash and all scripts in that directory are executed when a new shell starts. To run the script manually, e.g. if you have installed using SDKMAN!

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

$ spring <HIT TAB HERE>

grab help jar run test version

|  |
| --- |
| [Note] |
| If you install Spring Boot CLI using Homebrew or MacPorts, the command-line completion scripts are automatically registered with your shell. |

**10.2.6 Quick start Spring CLI example**

Here’s a really simple web application that you can use to test your installation. Create a file called app.groovy:

*@RestController*

**class** ThisWillActuallyRun {

*@RequestMapping("/")*

String home() {

"Hello World!"

}

}

Then simply run it from a shell:

$ spring run app.groovy

|  |
| --- |
| [Note] |
| It will take some time when you first run the application as dependencies are downloaded. Subsequent runs will be much quicker. |

Open [localhost:8080](http://localhost:8080/) in your favorite web browser and you should see the following output:

Hello World!

**10.3 Upgrading from an earlier version of Spring Boot**

If you are upgrading from an earlier release of Spring Boot check the “release notes” hosted on the [project wiki](https://github.com/spring-projects/spring-boot/wiki). You’ll find upgrade instructions along with a list of “new and noteworthy” features for each release.

To upgrade an existing CLI installation use the appropriate package manager command (for example brew upgrade) or, if you manually installed the CLI, follow the [standard instructions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-manual-cli-installation) remembering to update your PATH environment variable to remove any older references.

**11. Developing your first Spring Boot application**

Let’s develop a simple “Hello World!” web application in Java that highlights some of Spring Boot’s key features. We’ll use Maven to build this project since most IDEs support it.

|  |
| --- |
| [Tip] |
| The [spring.io](https://spring.io/) web site contains many “Getting Started” guides that use Spring Boot. If you’re looking to solve a specific problem; check there first.  You can shortcut the steps below by going to [start.spring.io](https://start.spring.io/) and choosing the web starter from the dependencies searcher. This will automatically generate a new project structure so that you can [start coding right away](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-code). Check the [documentation for more details](https://github.com/spring-io/initializr). |

Before we begin, open a terminal to check that you have valid versions of Java and Maven installed.

$ 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] |
| This sample needs to be created in its own folder. Subsequent instructions assume that you have created a suitable folder and that it is your “current directory”. |

**11.1 Creating the POM**

We need to start by creating a Maven pom.xml file. The pom.xml is the recipe（指导指令） that will be used to build your project. Open your favorite text editor and add the following:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 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>1.5.7.RELEASE</version>

</parent>

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

</project>

This should give you a working build, you can test it out by running mvn package (you can ignore the “jar will be empty - no content was marked for inclusion!” warning for now).

|  |
| --- |
| [Note] |
| At this point you could import the project into an IDE (most modern Java IDE’s include built-in support for Maven). For simplicity, we will continue to use a plain text editor for this example. |

**11.2 Adding classpath dependencies**

Spring Boot provides a number of “Starters” that make easy to add jars to your classpath. Our sample application has already used spring-boot-starter-parent in the parent section of the POM. The spring-boot-starter-parent is a special starter that provides useful Maven defaults. It also provides a [dependency-management](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-dependency-management) section so that you can omit version tags for “blessed” dependencies.

Other “Starters” simply provide dependencies that you are likely to need when developing a specific type of application. Since we are developing a web application, we will add a spring-boot-starter-web dependency — but before that, let’s look at what we currently have.

$ mvn dependency:tree

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

The mvn dependency:tree command prints a tree representation of your project dependencies. You can see that spring-boot-starter-parent provides no dependencies by itself. Let’s edit our pom.xml and add the spring-boot-starter-web dependency just below the parent section:

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

If you run mvn dependency:tree again, you will see that there are now a number of additional dependencies, including the Tomcat web server and Spring Boot itself.

**11.3 Writing the code**

To finish our application we need to create a single Java file. Maven will compile sources from src/main/java by default so you need to create that folder structure, then add a file named src/main/java/Example.java:

**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);

}

}

Although there isn’t much code here, quite a lot is going on. Let’s step through the important parts.

**11.3.1 The @RestController and @RequestMapping annotations**

The first annotation on our Example class is @RestController. This is known as a *stereotype(老规矩的)* annotation. It provides hints for people reading the code, and for Spring, that the class plays a specific role（该类扮演了一个特殊的角色）. In this case, our class is a web @Controller so Spring will consider it when handling incoming web requests.

The @RequestMapping annotation provides “routing” information（提供了路由信息）. It is telling Spring that any HTTP request with the path “/” should be mapped to the home method. The@RestController annotation tells Spring to render the resulting string directly back to the caller（RestController注解告诉spring渲染方式，即将结果字符串直接返回给访问者，类似spring中的responsebody注解）.

|  |
| --- |
| [Tip] |
| The @RestController and @RequestMapping annotations are Spring MVC annotations (they are not specific to Spring Boot). See the [MVC section](http://docs.spring.io/spring/docs/4.3.11.RELEASE/spring-framework-reference/htmlsingle#mvc) in the Spring Reference Documentation for more details. |

**11.3.2 The @EnableAutoConfiguration annotation**

The second class-level annotation is @EnableAutoConfiguration.(该注解告诉spring boot猜测我们的需求，自动进行配置-可能参考我们的依赖，如我们依赖了一个web启动则会自动部署为web应用) This annotation tells Spring Boot to “guess” how you will want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration will assume that you are developing a web application and setup Spring accordingly.

**Starters and Auto-Configuration**

Auto-configuration is designed to work well with “Starters”, but the two concepts are not directly tied. You are free to pick-and-choose jar dependencies outside of the starters and Spring Boot will still do its best to auto-configure your application.

**11.3.3 The “main” method**

The final part of our application is the main method. This is just a standard method that follows the Java convention for an application entry point. Our main method delegates to Spring Boot’s SpringApplication class by calling run.（通过调用run方法，将我们的main方法委托给了spring boot的SpringApplication类） SpringApplication will bootstrap our application, starting Spring which will in turn start the auto-configured Tomcat web server. We need to pass Example.class as an argument to the run method to tell SpringApplication which is the primary Spring component. The args array is also passed through to expose any command-line arguments.

**11.4 Running the example**

At this point our application should work. Since we have used the spring-boot-starter-parent POM we have a useful run goal that we can use to start the application. Type mvn spring-boot:run from the root project directory to start the application:

$ mvn spring-boot:run

. \_\_\_\_ \_ \_\_ \_ \_

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

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

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

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

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

:: Spring Boot :: (v1.5.7.RELEASE)

....... . . .

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

....... . . .

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

If you open a web browser to [localhost:8080](http://localhost:8080/) you should see the following output:

Hello World!

To gracefully exit the application hit ctrl-c.

**11.5 Creating an executable jar**

Let’s finish our example by creating a completely self-contained executable jar file that we could run in production. Executable jars (sometimes called “fat jars”) are archives containing your compiled classes along with all of the jar dependencies that your code needs to run（jar包包含了你的代码和依赖的代码）.

**Executable jars and Java**

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.

To solve this problem, many developers use “uber” jars. An uber jar simply packages all classes, from all jars, into a single archive. The problem with this approach 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar) and allows you to actually nest jars directly.

To create an executable jar we need to add the spring-boot-maven-plugin to our pom.xml. Insert the following lines just below the dependencies section:

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

|  |
| --- |
| [Note] |
| The spring-boot-starter-parent POM includes <executions> configuration to bind the repackage goal. If you are not using the parent POM you will need to declare this configuration yourself. See the [plugin documentation](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/usage.html) for details. |

Save your pom.xml and run mvn package from the command line:

$ 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:1.5.7.RELEASE:repackage (default) @ myproject ---

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

[INFO] BUILD SUCCESS

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

If you look in the target directory you should see myproject-0.0.1-SNAPSHOT.jar. The file should be around 10 MB in size. If you want to peek inside, you can use jar tvf（该命令可以查看包里面的文件）:

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

You should also see a much smaller file named myproject-0.0.1-SNAPSHOT.jar.original in the target directory. This is the original jar file that Maven created before it was repackaged by Spring Boot.

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

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

. \_\_\_\_ \_ \_\_ \_ \_

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

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

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

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

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

:: Spring Boot :: (v1.5.7.RELEASE)

....... . . .

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

....... . . .

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

As before, to gracefully exit the application hit ctrl-c.（使用ctrl-c可以优雅的退出应用）

**12. What to read next**

Hopefully this section has provided you with some of the Spring Boot basics, and got you on your way to writing your own applications. If you’re a task-oriented type（任务导向型） of developer you might want to jump over to [spring.io](https://spring.io/) and check out some of the [getting started](https://spring.io/guides/) guides that solve specific “How do I do that with Spring” problems; we also have Spring Boot-specific [*How-to*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto) reference documentation.

The [Spring Boot repository](https://github.com/spring-projects/spring-boot) has also a [bunch of samples](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples) you can run. The samples are independent of the rest of the code (that is you don’t need to build the rest to run or use the samples).

Otherwise, the next logical step is to read [*Part III, “Using Spring Boot”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot). If you’re really impatient, you could also jump ahead and read about [*Spring Boot features*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features).

**Part III. Using Spring Boot**

This section goes into more detail about how you should use Spring Boot. It covers topics such as build systems, auto-configuration and how to run your applications. We also cover some Spring Boot best practices. Although there is nothing particularly special about Spring Boot (it is just another library that you can consume), there are a few recommendations that, when followed, will make your development process just a little easier.

If you’re just starting out with Spring Boot, you should probably read the [*Getting Started*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started) guide before diving into this section.

**13. Build systems**

It is strongly recommended that you choose a build system（构建系统） that supports [*dependency management*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-dependency-management), and one that can consume artifacts published to the “Maven Central” repository. We would recommend that you choose Maven or Gradle. It is possible to get Spring Boot to work with other build systems (Ant for example), but they will not be particularly well supported.

**13.1 Dependency management**

Each release of Spring Boot provides a curated list of dependencies it supports. In practice, you do not need to provide a version for any of these dependencies in your build configuration as Spring Boot is managing that for you. When you upgrade Spring Boot itself, these dependencies will be upgraded as well in a consistent way.

|  |
| --- |
| [Note] |
| You can still specify a version and override Spring Boot’s recommendations if you feel that’s necessary. |

The curated list contains all the spring modules that you can use with Spring Boot as well as a refined list of third party libraries. The list is available as a standard [Bills of Materials (spring-boot-dependencies)](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-without-a-parent) and additional dedicated support for [Maven](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-maven-parent-pom) and [Gradle](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-dependency-management) are available as well.

|  |
| --- |
| [Warning] |
| Each release of Spring Boot is associated with a base version of the Spring Framework so we **highly** recommend you to not specify its version on your own. |

**13.2 Maven**

Maven users can inherit from the spring-boot-starter-parent project to obtain sensible defaults. The parent project provides the following features:

* Java 1.6 as the default compiler level.
* UTF-8 source encoding.
* A [Dependency Management section](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-dependency-management), allowing you to omit <version> tags for common dependencies, inherited from the spring-boot-dependencies POM（不需要指定版本则最好继承spring-boot-starter-parent）.
* Sensible [resource filtering](https://maven.apache.org/plugins/maven-resources-plugin/examples/filter.html).
* Sensible plugin configuration ([exec plugin](http://www.mojohaus.org/exec-maven-plugin/), [surefire](https://maven.apache.org/surefire/maven-surefire-plugin/), [Git commit ID](https://github.com/ktoso/maven-git-commit-id-plugin), [shade](https://maven.apache.org/plugins/maven-shade-plugin/)).
* （自动扫描配置文件）Sensible resource filtering for application.properties and application.yml including profile-specific files (e.g. application-foo.properties and application-foo.yml)

On the last point: since the default config files accept Spring style placeholders (${…​}) the Maven filtering is changed to use @[[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection) placeholders (you can override that with a Maven property resource.delimiter).

**13.2.1 Inheriting the starter parent**

To configure your project to inherit from the spring-boot-starter-parent simply set the parent:

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

<parent>

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

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

<version>1.5.7.RELEASE</version>

</parent>

|  |
| --- |
| [Note] |
| You should only need to specify the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number. |

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to upgrade to another Spring Data release train you’d add the following to your pom.xml.

<properties>

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

</properties>

|  |
| --- |
| [Tip] |
| Check the [spring-boot-dependencies pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-dependencies/pom.xml) for a list of supported properties. |

**13.2.2 Using Spring Boot without the parent POM**

Not everyone likes inheriting from the spring-boot-starter-parent POM. You may have your own corporate standard parent that you need to use, or you may just prefer to explicitly declare all your Maven configuration.

If you don’t want to use the spring-boot-starter-parent, you can still keep the benefit of the dependency management (but not the plugin management) by using a scope=import dependency:

<dependencyManagement>

<dependencies>

<dependency>

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

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

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

<version>1.5.7.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

That setup does not allow you to override individual dependencies using a property as explained above. To achieve the same result, you’d need to add an entry in thedependencyManagement of your project **before** the spring-boot-dependencies entry. For instance, to upgrade to another Spring Data release train you’d add the following to your pom.xml.

<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>1.5.7.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

|  |
| --- |
| [Note] |
| In the example above, we specify a *BOM* but any dependency type can be overridden that way. |

**13.2.3 Changing the Java version**

The spring-boot-starter-parent chooses fairly conservative Java compatibility. If you want to follow our recommendation and use a later Java version you can add a java.version property:

<properties>

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

</properties>

**13.2.4 Using the Spring Boot Maven plugin**

Spring Boot includes a [Maven plugin](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-maven-plugin) that can package the project as an executable jar. Add the plugin to your <plugins> section if you want to use it:

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

|  |
| --- |
| [Note] |
| If you use the Spring Boot starter parent pom, you only need to add the plugin, there is no need for to configure it unless you want to change the settings defined in the parent. |

**13.3 Gradle**

Gradle users can directly import ‘starters’ in their dependencies section. Unlike Maven, there is no “super parent” to import to share some configuration.

repositories {

jcenter() #gradle的仓库中心

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web:1.5.7.RELEASE")

}

The [spring-boot-gradle-plugin](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-plugin) is also available and provides tasks to create executable jars and run projects from source. It also provides [dependency management](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-dependency-management) that, among other capabilities, allows you to omit the version number for any dependencies that are managed by Spring Boot:

plugins {

id 'org.springframework.boot' version '1.5.7.RELEASE'

id 'java'

}

repositories {

jcenter()

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

**13.4 Ant**

It is possible to build a Spring Boot project using Apache Ant+Ivy. The spring-boot-antlib “AntLib” module is also available to help Ant create executable jars.

To declare dependencies a typical ivy.xml file will look something like this:

<ivy-module version="2.0">

<info organisation="org.springframework.boot" module="spring-boot-sample-ant" />

<configurations>

<conf name="compile" description="everything needed to compile this module" />

<conf name="runtime" extends="compile" description="everything needed to run this module" />

</configurations>

<dependencies>

<dependency org="org.springframework.boot" name="spring-boot-starter"

rev="${spring-boot.version}" conf="compile" />

</dependencies>

</ivy-module>

A typical build.xml will look like this:

<project

xmlns:ivy="antlib:org.apache.ivy.ant"

xmlns:spring-boot="antlib:org.springframework.boot.ant"

name="myapp" default="build">

<property name="spring-boot.version" value="1.3.0.BUILD-SNAPSHOT" />

<target name="resolve" description="--> retrieve dependencies with ivy">

<ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]" />

</target>

<target name="classpaths" depends="resolve">

<path id="compile.classpath">

<fileset dir="lib/compile" includes="\*.jar" />

</path>

</target>

<target name="init" depends="classpaths">

<mkdir dir="build/classes" />

</target>

<target name="compile" depends="init" description="compile">

<javac srcdir="src/main/java" destdir="build/classes" classpathref="compile.classpath" />

</target>

<target name="build" depends="compile">

<spring-boot:exejar destfile="build/myapp.jar" classes="build/classes">

<spring-boot:lib>

<fileset dir="lib/runtime" />

</spring-boot:lib>

</spring-boot:exejar>

</target>

</project>

|  |
| --- |
| [Tip] |
| See the [*Section 84.10, “Build an executable archive from Ant without using spring-boot-antlib”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build-an-executable-archive-with-ant) “How-to” if you don’t want to use the spring-boot-antlib module. |

**13.5 Starters**

Starter插件包

Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, just include the spring-boot-starter-data-jpa dependency in your project, and you are good to go.

The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

**What’s in a name**

All **official** starters follow a similar naming pattern; spring-boot-starter-\*, where \* is a particular type of application. This naming structure is intended to help when you need to find a starter. The Maven integration in many IDEs allow you to search dependencies by name. For example, with the appropriate Eclipse or STS plugin installed, you can simply hit ctrl-space in the POM editor and type “spring-boot-starter” for a complete list.

As explained in the [Creating your own starter](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter) section, third party starters should not start with spring-boot as it is reserved for official Spring Boot artifacts. A third-party starter for acme will be typically named acme-spring-boot-starter.

The following application starters are provided by Spring Boot under the org.springframework.boot group:

**Table 13.1. Spring Boot application starters**

| **Name** | **Description** | **Pom** |
| --- | --- | --- |
| spring-boot-starter | Core starter, including auto-configuration support, logging and YAML（核心starter，包含自动配置启动和日志） | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter/pom.xml) |
| spring-boot-starter-activemq | Starter for JMS messaging using Apache ActiveMQ（使用apache的acativemq消息队列的JMS消息） | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-activemq/pom.xml) |
| spring-boot-starter-amqp | Starter for using Spring AMQP and Rabbit MQ  启动spring amqp和rabbit mq | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-amqp/pom.xml) |
| spring-boot-starter-aop | Starter for aspect-oriented programming with Spring AOP and AspectJ | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-aop/pom.xml) |
| spring-boot-starter-artemis | Starter for JMS messaging using Apache Artemis | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-artemis/pom.xml) |
| spring-boot-starter-batch | Starter for using Spring Batch | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-batch/pom.xml) |
| spring-boot-starter-cache | Starter for using Spring Framework’s caching support | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-cache/pom.xml) |
| 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/v1.5.7.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/v1.5.7.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/v1.5.7.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/v1.5.7.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/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-data-gemfire/pom.xml) |
| spring-boot-starter-data-jpa | Starter for using Spring Data JPA with Hibernate | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-data-jpa/pom.xml) |
| spring-boot-starter-data-ldap | Starter for using Spring Data LDAP | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-data-ldap/pom.xml) |
| 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/v1.5.7.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/v1.5.7.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/v1.5.7.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/v1.5.7.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/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-data-solr/pom.xml) |
| spring-boot-starter-freemarker | Starter for building MVC web applications using FreeMarker views | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-groovy-templates/pom.xml) |
| spring-boot-starter-hateoas | Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-hateoas/pom.xml) |
| spring-boot-starter-integration | Starter for using Spring Integration | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-integration/pom.xml) |
| spring-boot-starter-jdbc | Starter for using JDBC with the Tomcat JDBC connection pool | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-jdbc/pom.xml) |
| spring-boot-starter-jersey | Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to [spring-boot-starter-web](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-web) | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-jersey/pom.xml) |
| spring-boot-starter-jooq | Starter for using jOOQ to access SQL databases. An alternative to [spring-boot-starter-data-jpa](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-data-jpa) or [spring-boot-starter-jdbc](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-jdbc) | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-jooq/pom.xml) |
| spring-boot-starter-jta-atomikos | Starter for JTA transactions using Atomikos | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-jta-atomikos/pom.xml) |
| spring-boot-starter-jta-bitronix | Starter for JTA transactions using Bitronix | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-jta-bitronix/pom.xml) |
| spring-boot-starter-jta-narayana | Spring Boot Narayana JTA Starter | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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/v1.5.7.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/v1.5.7.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/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-mustache/pom.xml) |
| spring-boot-starter-security | Starter for using Spring Security | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-security/pom.xml) |
| spring-boot-starter-social-facebook | Starter for using Spring Social Facebook | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-social-facebook/pom.xml) |
| spring-boot-starter-social-linkedin | Stater for using Spring Social LinkedIn | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-social-linkedin/pom.xml) |
| spring-boot-starter-social-twitter | Starter for using Spring Social Twitter | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-social-twitter/pom.xml) |
| spring-boot-starter-test | Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-test/pom.xml) |
| spring-boot-starter-thymeleaf | Starter for building MVC web applications using Thymeleaf views | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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/v1.5.7.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/v1.5.7.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/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-web-services/pom.xml) |
| spring-boot-starter-websocket | Starter for building WebSocket applications using Spring Framework’s WebSocket support | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-websocket/pom.xml) |

In addition to the application starters, the following starters can be used to add [*production ready*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready) features（当项目作为产品发布或者被部署到生产环境下可以使用一下启动项插件）:

**Table 13.2. Spring Boot production starters**

| **Name** | **Description** | **Pom** |
| --- | --- | --- |
| spring-boot-starter-actuator | Starter for using Spring Boot’s Actuator which provides production ready features to help you monitor and manage your application  该插件能够监控和管理你的应用 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-actuator/pom.xml) |
| spring-boot-starter-remote-shell | Starter for using the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5  该启动项可以让你通过ssh使用CRaSH的远程终端监控和管理你的应用 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-remote-shell/pom.xml) |

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

如果你想排除掉某些特殊的技术包的话，spring boot的某些starter能够派上用场

**Table 13.3. Spring Boot technical starters**

Spring boot的技术的starters

| **Name** | **Description** | **Pom** |
| --- | --- | --- |
| spring-boot-starter-jetty | Starter for using Jetty as the embedded servlet container. An alternative to [spring-boot-starter-tomcat](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-tomcat)  使用jetty作为内嵌的servlet容器，也可以使用tomcat的starter即[spring-boot-starter-tomcat](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-tomcat) | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-jetty/pom.xml) |
| spring-boot-starter-log4j2 | Starter for using Log4j2 for logging. An alternative to [spring-boot-starter-logging](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-logging)  使用log4j做日志记载，也可以选择[spring-boot-starter-logging](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-logging) | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-log4j2/pom.xml) |
| spring-boot-starter-logging | Starter for logging using Logback. Default logging starter  使用logback做日志记录，这个是默认的日志starter。 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-logging/pom.xml) |
| spring-boot-starter-tomcat | Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by [spring-boot-starter-web](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-web)  使用tomcat作为嵌入的servlet容器，当使用[spring-boot-starter-web](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-web) 时默认使用的就是内嵌tomcat的容器 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-tomcat/pom.xml) |
| spring-boot-starter-undertow | Starter for using Undertow as the embedded servlet container. An alternative to [spring-boot-starter-tomcat](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#spring-boot-starter-tomcat)  使用Undertow作为servlet容器，也可使用tomcat嵌入来代替 | [Pom](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-starters/spring-boot-starter-undertow/pom.xml) |

|  |
| --- |
| [Tip] |
| For a list of additional community contributed starters, see the [README file](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/README.adoc) in the spring-boot-starters module on GitHub. |

**14. Structuring your code**

Spring Boot does not require any specific code layout to work（spring对代码结构没有要求，你可以使用任何的代码结构）, however, there are some best practices that help（以下是一些较好的实践方式，能够提供更多的帮助）.

**14.1 Using the “default” package**

When a class doesn’t include a package declaration it is considered to be in the “default package”. The use of the “default package” is generally discouraged, and should be avoided. It can cause particular problems for Spring Boot applications that use @ComponentScan, @EntityScan or @SpringBootApplication annotations, since every class from every jar, will be read.

最好不要使用默认的包（即当不建立包而直接编写代码时实际上是存在默认的包的），最好将代码放在指定的包里面，否则spring boot的某些注解工作可能出现异常

|  |
| --- |
| [Tip] |
| We recommend that you follow Java’s recommended package naming conventions and use a reversed domain name (for example, com.example.project).（建议使用域名反写的形式作为包名，这也是java编码的惯例） |

**14.2 Locating the main application class**

We generally recommend that you locate your main application class in a root package above other classes.

我们一般建议将你的mian应用放在根包，在其他的上层。应为spring boot默认扫描main方法所在的包即其下的子包。

The @EnableAutoConfiguration annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items（通常，将EnableAutoConfiguration注解放在你的main类中，这样它会自动的为items搜索特定的包）. For example, if you are writing a JPA application, the package of the@EnableAutoConfiguration annotated class will be used to search for @Entity items.

Using a root package also allows the @ComponentScan annotation to be used without needing to specify a basePackage attribute（如果使用了ComponentScan 注解的话，就不需要指定backPackage 属性了）. You can also use the@SpringBootApplication annotation if your main class is in the root package（如果你的main方法在root package中，你也可以使用SpringBootApplication 注解）.

Here is a typical layout:

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.

Application.java文件位于根包中，在里面定义main方法，而且设置了基本的@Configuration 注解。

**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);

}

}

**15. Configuration classes**

Spring Boot favors Java-based configuration. Although it is possible to call SpringApplication.run() with an XML source, we generally recommend that your primary source is a @Configuration class（强烈建议主要的源文件被@Configuration 注解修饰）. Usually the class that defines the main method is also a good candidate as the primary @Configuration.

可以在主要的源文件中使用@Configuration 注解，当时通常也建议使用java基本的xml配置。

|  |
| --- |
| [Tip] |
| Many Spring configuration examples have been published on the Internet that use XML configuration. Always try to use the equivalent Java-based configuration if possible. Searching for Enable\* annotations can be a good starting point. |

**15.1 Importing additional configuration classes**

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

导入额外的配置类：你不需要为每个单独的类都是用@Configuration 注解，可以将@Import 用于导入的配置类。还可以：使用@ComponentScan 注解以便自动扫描选取spring 组件，包括@Configuration 注释的类。

**15.2 Importing XML configuration**

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

导入xml配置：如果确定要使用xml配置的话，我们将以你将@Configuration 注解到应用启动类上。然后你就可以使用@ImportResource 来注解导入xml配置文件

**16. Auto-configuration**

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

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

Spring auto-configuration组件自动配置你的spring应用，基于

你添加的依赖jar包。例如，如果你依赖了HSQLDB ，并且你没有手动配置任何的数据库连接beans,使用了auto-configuration后将会自动配置一个内存数据库。

这需要你为@Configuration 修饰的类添加额外的@EnableAutoConfiguration 注解或者@SpringBootApplication 注解。

|  |
| --- |
| [Tip] |
| You should only ever add one @EnableAutoConfiguration annotation. We generally recommend that you add it to your primary @Configuration class. |

**16.1 Gradually replacing auto-configuration**

Auto-configuration is noninvasive, at any point you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own DataSource bean, the default embedded database support will back away.

If you need to find out what auto-configuration is currently being applied, and why, start your application with the --debug switch. This will enable debug logs for a selection of core loggers and log an auto-configuration report to the console.

逐渐的替代auto-configuration. Auto-configuration的配置是非侵入性的-即可以自由的移除和添加。在任何时候你都可以使用你自己的配置代替auto-configuration.

例如，如果你添加了字节的数据库源的bean，那么默认嵌入的数据库支持将会自动的移除。

如果你想查看程序的哪些部分使用了auto-configuration以及为什么，以—debug模式启动你的应用，这些信息将会输出在log中。并且会将auto-configuration的信息打印到控制台中。

**16.2 Disabling specific auto-configuration**

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

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

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

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

*@Configuration*

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

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

}

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

使用exclude 属性禁用某些项的自动配置

|  |
| --- |
| [Tip] |
| You can define exclusions both at the annotation level and using the property. |

**17. Spring Beans and dependency injection**

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. For simplicity, we often find that using @ComponentScan to find your beans, in combination with @Autowired constructor injection works well.

If you structure your code as suggested above (locating your application class in a root package), you can add @ComponentScan without any arguments. All of your application components (@Component, @Service, @Repository, @Controller etc.) will be automatically registered as Spring Beans.

Here is an example @Service Bean that uses constructor injection to obtain a required 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;

}

*// ...*

}

And if a bean has one constructor, you can omit the @Autowired.

*@Service*

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

**private** **final** RiskAssessor riskAssessor;

**public** DatabaseAccountService(RiskAssessor riskAssessor) {

**this**.riskAssessor = riskAssessor;

}

*// ...*

}

Spring 的beans和依赖注入：componentScan会自动扫描会找到可用的spring beans, Autowired 会为你自动注入需要的依赖。

如果你的项目结构像上面示例额那样（main方法所在的类在根包中）。你可将@ComponentScan注解修饰main方法所在的类，而且不需还要任何的参数。所有的你的组件@Component, @Service, @Repository, @Controller 将会被自动注册为spring的bean。

这里是一个@Service 的例子，该类使用构造方法注入一个依赖对象

如果一个类有构造方法，可以在构造方法里面 将依赖对象注入，不需要使用@Autowired。况且可以将依赖项声明为final，而自动注入不能将依赖声明为final。

|  |
| --- |
| [Tip] |
|
|
| Notice how using constructor injection allows the riskAssessor field to be marked as final, indicating that it cannot be subsequently changed. |

**18. Using the @SpringBootApplication annotation**

Many Spring Boot developers always have their main class annotated with @Configuration, @EnableAutoConfiguration and @ComponentScan. Since these annotations are so frequently used together (especially if you follow the [best practices](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-structuring-your-code) above), Spring Boot provides a convenient @SpringBootApplicationalternative.

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

**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);

}

}

使用@SpringBootApplication 注解，许多的开发者给他们的main方法所在的类添加@Configuration, @EnableAutoConfiguration and @ComponentScan 这些注解，因为这些注解如此被频繁使用，所以spring boot提供了一个可选择的

@SpringBootApplication 注解。该注解与其他的几个注解具有相同的默认属性。

|  |
| --- |
| [Note] |
| @SpringBootApplication also provides aliases to customize the attributes of @EnableAutoConfiguration and @ComponentScan. |

**19. Running your application**

One of the biggest advantages of packaging your application as jar and using an embedded HTTP server is that you can run your application as you would any other. Debugging Spring Boot applications is also easy; you don’t need any special IDE plugins or extensions.

|  |
| --- |
| [Note] |
| This section only covers jar based packaging, If you choose to package your application as a war file you should refer to your server and IDE documentation. |

运行应用：使用一个内嵌的HTTP服务器，并将你的应用打包成jar包的一个优势是你可以向运行其他应用那样运行该应用。调试spring boot应用也是非常的容易；不需要安装任何特殊的ide插件或者扩展。

打包成jar包后，可以使用java运行：java –jar your-app.jar

**19.1 Running from an IDE**

You can run a Spring Boot application from your IDE as a simple Java application, however, first you will need to import your project. Import steps will vary depending on your IDE and build system. Most IDEs can import Maven projects directly, for example Eclipse users can select Import…​ → Existing Maven Projects from the File menu.

If you can’t directly import your project into your IDE, you may be able to generate IDE metadata using a build plugin. Maven includes plugins for [Eclipse](https://maven.apache.org/plugins/maven-eclipse-plugin/) and [IDEA](https://maven.apache.org/plugins/maven-idea-plugin/); Gradle offers plugins for [various IDEs](https://docs.gradle.org/2.14.1/userguide/userguide.html).

|  |
| --- |
| [Tip] |
| If you accidentally run a web application twice you will see a “Port already in use” error. STS users can use the Relaunch button rather than Run to ensure that any existing instance is closed. |

可以在ide里面直接运行你的spring boot应用，就像启动其他的简单的java应用一样。如果你不能直接导入你的项目到ide的话，可使用构建插件生成ide的metadata以便导入。

**19.2 Running as a packaged application**

If you use the Spring Boot Maven or Gradle plugins to create an executable jar you can run your application using java -jar. For example:

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

It is also possible to run a packaged application with remote debugging support enabled. This allows you to attach a debugger to your packaged application:

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

-jar target/myproject-0.0.1-SNAPSHOT.jar

以一个包应用的形式运行你的应用：

如果使用了构建插件生成了可执行的jar包，那么你可以使用java –jar命令来运行你的应用。

你也可以运行一个带有远程调试的包，这将允许你附加一个调试器到包应用中：

如：

**19.3 Using the Maven plugin**

The Spring Boot Maven plugin includes a run goal which can be used to quickly compile and run your application. Applications run in an exploded form just like in your IDE.

$ mvn spring-boot:run

You might also want to use the useful operating system environment variable:

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

使用maven插件运行你的应用-这需要你的应用是由maven构建的：

Spring boot的插件包含run命令，可以快速的编译和运行你的应用。应用以exploded form的形式运行，就像你的ide一样。

**19.4 Using the Gradle plugin**

The Spring Boot Gradle plugin also includes a bootRun task which can be used to run your application in an exploded form. The bootRun task is added whenever you import the spring-boot-gradle-plugin:

$ gradle bootRun

You might also want to use this useful operating system environment variable:

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

类似maven，gradle插件内置了一个bootRun的task，这个task也会编译并运行你的应用。

在项目根目录执行：gradle bootRun

**19.5 Hot swapping**

Since Spring Boot applications are just plain Java applications, JVM hot-swapping should work out of the box. JVM hot swapping is somewhat limited with the bytecode that it can replace, for a more complete solution [JRebel](http://zeroturnaround.com/software/jrebel/) or the [Spring Loaded](https://github.com/spring-projects/spring-loaded) project can be used. The spring-boot-devtools module also includes support for quick application restarts.

See the [Chapter 20, *Developer tools*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools) section below and the [Hot swapping “How-to”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-hotswapping) for details.

Spring boot应用仅仅是一般的java应用，所以jvm hot-swapping会工作的很好

**20. Developer tools**

Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The spring-boot-devtools module can be included in any project to provide additional development-time features. To include devtools support, simply add the module dependency to your build:

**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] |
| Developer tools are automatically disabled when running a fully packaged application. If your application is launched using java -jar or if it’s started using a special classloader, then it is considered a “production application”. Flagging the dependency as optional is a best practice that prevents devtools from being transitively applied to other modules using your project. Gradle does not support optional dependencies out-of-the-box so you may want to have a look to the [propdeps-plugin](https://github.com/spring-projects/gradle-plugins/tree/master/propdeps-plugin) in the meantime. |
| [Tip] |
| repackaged archives do not contain devtools by default. If you want to use [certain remote devtools feature](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-remote), you’ll need to disable the excludeDevtoolsbuild property to include it. The property is supported with both the Maven and Gradle plugins. | |

**20.1 Property defaults**

Several of the libraries supported by Spring Boot use caches to improve performance. For example, [template engines](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-template-engines) will cache compiled templates to avoid repeatedly parsing template files. Also, Spring MVC can add HTTP caching headers to responses when serving static resources.

Whilst caching is very beneficial in production, it can be counter productive during development, preventing you from seeing the changes you just made in your application. For this reason, spring-boot-devtools will disable those caching options by default.

Cache options are usually configured by settings in your application.properties file. For example, Thymeleaf offers the spring.thymeleaf.cache property. Rather than needing to set these properties manually, the spring-boot-devtools module will automatically apply sensible development-time configuration.

为了提高性能，spring boot使用的许多依赖库使用啦缓存。例如，模板引擎会缓存编译好的文件已避免重复的解析模板文件。而且，当spring mvc提供静态资源服务的时候也会提供http缓存头设置。

//======20170927

|  |
| --- |
| [Tip] |
| For a complete list of the properties that are applied see [DevToolsPropertyDefaultsPostProcessor](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-devtools/src/main/java/org/springframework/boot/devtools/env/DevToolsPropertyDefaultsPostProcessor.java). |

**20.2 Automatic restart**

Applications that use spring-boot-devtools will automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder will be monitored for changes. Note that certain resources such as static assets and view templates [do not need to restart the application](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-exclude).

**Triggering a restart**

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using. In Eclipse, saving a modified file will cause the classpath to be updated and trigger a restart. In IntelliJ IDEA, building the project (Build -> Make Project) will have the same effect.

|  |
| --- |
| [Note] |
| You can also start your application via the supported build plugins (i.e. Maven and Gradle) as long as forking is enabled since DevTools need an isolated application classloader to operate properly. Gradle and Maven do that by default when they detect DevTools on the classpath. |
| [Tip] |
| Automatic restart works very well when used with LiveReload. [See below](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-livereload) for details. If you use JRebel automatic restarts will be disabled in favor of dynamic class reloading. Other devtools features (such as LiveReload and property overrides) can still be used. | |

|  |
| --- |
| [Note] |
| DevTools relies on the application context’s shutdown hook to close it during a restart. It will not work correctly if you have disabled the shutdown hook (SpringApplication.setRegisterShutdownHook(false)). | |
| [Note] | |
| When deciding if an entry on the classpath should trigger a restart when it changes, DevTools automatically ignores projects named spring-boot, spring-boot-devtools, spring-boot-autoconfigure, spring-boot-actuator, and spring-boot-starter. | |

|  |
| --- |
| [Note] |
| DevTools needs to customize the ResourceLoader used by the ApplicationContext: if your application provides one already, it is going to be wrapped. Direct override of the getResource method on the ApplicationContext is not supported. |

**Restart vs Reload**

The restart technology provided by Spring Boot works by using two classloaders. Classes that don’t change (for example, those from third-party jars) are loaded into a *base* classloader. Classes that you’re actively developing are loaded into a *restart* classloader. When the application is restarted, the *restart* classloader is thrown away and a new one is created. This approach means that application restarts are typically much faster than “cold starts” since the *base* classloader is already available and populated.

If you find that restarts aren’t quick enough for your applications, or you encounter classloading issues, you could consider reloading technologies such as [JRebel](http://zeroturnaround.com/software/jrebel/)from ZeroTurnaround. These work by rewriting classes as they are loaded to make them more amenable to reloading. [Spring Loaded](https://github.com/spring-projects/spring-loaded) provides another option, however it doesn’t support as many frameworks and it isn’t commercially supported.

**20.2.1 Excluding resources**

Certain resources don’t necessarily need to trigger a restart when they are changed. For example, Thymeleaf templates can just be edited in-place. By default changing resources in /META-INF/maven, /META-INF/resources, /resources, /static, /public or /templates will not trigger a restart but will trigger a [live reload](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-livereload). If you want to customize these exclusions you can use the spring.devtools.restart.exclude property. For example, to exclude only /static and /public you would set the following:

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

|  |
| --- |
| [Tip] |
| if you want to keep those defaults and *add* additional exclusions, use the spring.devtools.restart.additional-exclude property instead. |

**20.2.2 Watching additional paths**

You may want your application to be restarted or reloaded when you make changes to files that are not on the classpath. To do so, use thespring.devtools.restart.additional-paths property to configure additional paths to watch for changes. You can use the spring.devtools.restart.excludeproperty [described above](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-exclude) to control whether changes beneath the additional paths will trigger a full restart or just a [live reload](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-livereload).

**20.2.3 Disabling restart**

If you don’t want to use the restart feature you can disable it using the spring.devtools.restart.enabled property. In most cases you can set this in yourapplication.properties (this will still initialize the restart classloader but it won’t watch for file changes).

If you need to *completely* disable restart support, for example, because it doesn’t work with a specific library, you need to set a System property before callingSpringApplication.run(…​). For example:

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

System.setProperty("spring.devtools.restart.enabled", "false");

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

}

**20.2.4 Using a trigger file**

If you work with an IDE that continuously compiles changed files, you might prefer to trigger restarts only at specific times. To do this you can use a “trigger file”, which is a special file that must be modified when you want to actually trigger a restart check. Changing the file only triggers the check and the restart will only occur if Devtools has detected it has to do something. The trigger file could be updated manually, or via an IDE plugin.

To use a trigger file use the spring.devtools.restart.trigger-file property.

|  |
| --- |
| [Tip] |
| You might want to set spring.devtools.restart.trigger-file as a [global setting](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-globalsettings) so that all your projects behave in the same way. |

**20.2.5 Customizing the restart classloader**

As described in the [Restart vs Reload](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-spring-boot-restart-vs-reload) section above, restart functionality is implemented by using two classloaders. For most applications this approach works well, however, sometimes it can cause classloading issues.

By default, any open project in your IDE will be loaded using the “restart” classloader, and any regular .jar file will be loaded using the “base” classloader. If you work on a multi-module project, and not each module is imported into your IDE, you may need to customize things. To do this you can create a META-INF/spring-devtools.properties file.

The spring-devtools.properties file can contain restart.exclude. and restart.include. prefixed properties. The include elements are items that should be pulled up into the “restart” classloader, and the exclude elements are items that should be pushed down into the “base” classloader. The value of the property is a regex pattern that will be applied to the classpath.

For example:

restart.exclude.companycommonlibs=/mycorp-common-[\\w-]+\.jar

restart.include.projectcommon=/mycorp-myproj-[\\w-]+\.jar

|  |
| --- |
| [Note] |
| All property keys must be unique. As long as a property starts with restart.include. or restart.exclude. it will be considered. |
| [Tip] |
| All META-INF/spring-devtools.properties from the classpath will be loaded. You can package files inside your project, or in the libraries that the project consumes. | |

**20.2.6 Known limitations**

Restart functionality does not work well with objects that are deserialized using a standard ObjectInputStream. If you need to deserialize data, you may need to use Spring’s ConfigurableObjectInputStream in combination with Thread.currentThread().getContextClassLoader().

Unfortunately, several third-party libraries deserialize without considering the context classloader. If you find such a problem, you will need to request a fix with the original authors.

**20.3 LiveReload**

The spring-boot-devtools module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload browser extensions are freely available for Chrome, Firefox and Safari from [livereload.com](http://livereload.com/extensions/).

If you don’t want to start the LiveReload server when your application runs you can set the spring.devtools.livereload.enabled property to false.

|  |
| --- |
| [Note] |
| You can only run one LiveReload server at a time. Before starting your application, ensure that no other LiveReload servers are running. If you start multiple applications from your IDE, only the first will have LiveReload support. |

**20.4 Global settings**

You can configure global devtools settings by adding a file named .spring-boot-devtools.properties to your $HOME folder (note that the filename starts with “.”). Any properties added to this file will apply to *all* Spring Boot applications on your machine that use devtools. For example, to configure restart to always use a [trigger file](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-triggerfile), you would add the following:

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

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

**20.5 Remote applications**

The Spring Boot developer tools are not just limited to local development. You can also use several features when running applications remotely. Remote support is opt-in, to enable it you need to make sure that devtools is included in the repackaged archive:

<build>

<plugins>

<plugin>

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

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

<configuration>

<excludeDevtools>false</excludeDevtools>

</configuration>

</plugin>

</plugins>

</build>

Then you need to set a spring.devtools.remote.secret property, for example:

spring.devtools.remote.secret=mysecret

|  |
| --- |
| [Warning] |
| Enabling spring-boot-devtools on a remote application is a security risk. You should never enable support on a production deployment. |

Remote devtools support is provided in two parts; there is a server side endpoint that accepts connections, and a client application that you run in your IDE. The server component is automatically enabled when the spring.devtools.remote.secret property is set. The client component must be launched manually.

**20.5.1 Running the remote client application**

The remote client application is designed to be run from within your IDE. You need to run org.springframework.boot.devtools.RemoteSpringApplication using the same classpath as the remote project that you’re connecting to. The *non-option* argument passed to the application should be the remote URL that you are connecting to.

For example, if you are using Eclipse or STS, and you have a project named my-app that you’ve deployed to Cloud Foundry, you would do the following:

* Select Run Configurations…​ from the Run menu.
* Create a new Java Application “launch configuration”.
* Browse for the my-app project.
* Use org.springframework.boot.devtools.RemoteSpringApplication as the main class.
* Add https://myapp.cfapps.io to the Program arguments (or whatever your remote URL is).

A running remote client will look like this:

. \_\_\_\_ \_ \_\_ \_ \_

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

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

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

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

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

:: Spring Boot Remote :: 1.5.7.RELEASE

2015-06-10 18:25:06.632 INFO 14938 --- [ main] o.s.b.devtools.RemoteSpringApplication : 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.spring[[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection)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://'.

2015-06-10 18:25:07.074 INFO 14938 --- [ main] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729

2015-06-10 18:25:07.130 INFO 14938 --- [ main] o.s.b.devtools.RemoteSpringApplication : Started RemoteSpringApplication in 0.74 seconds (JVM running for 1.105)

|  |
| --- |
| [Note] |
| Because the remote client is using the same classpath as the real application it can directly read application properties. This is how the spring.devtools.remote.secret property is read and passed to the server for authentication. |
| [Tip] |
| It’s always advisable to use https:// as the connection protocol so that traffic is encrypted and passwords cannot be intercepted. | |

|  |
| --- |
| [Tip] |
| If you need to use a proxy to access the remote application, configure the spring.devtools.remote.proxy.host and spring.devtools.remote.proxy.port properties. |

**20.5.2 Remote update**

The remote client will monitor your application classpath for changes in the same way as the [local restart](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart). Any updated resource will be pushed to the remote application and *(if required)* trigger a restart. This can be quite helpful if you are iterating on a feature that uses a cloud service that you don’t have locally. Generally remote updates and restarts are much quicker than a full rebuild and deploy cycle.

|  |
| --- |
| [Note] |
| Files are only monitored when the remote client is running. If you change a file before starting the remote client, it won’t be pushed to the remote server. |

**20.5.3 Remote debug tunnel**

Java remote debugging is useful when diagnosing issues on a remote application. Unfortunately, it’s not always possible to enable remote debugging when your application is deployed outside of your data center. Remote debugging can also be tricky to setup if you are using a container based technology such as Docker.

To help work around these limitations, devtools supports tunneling of remote debug traffic over HTTP. The remote client provides a local server on port 8000 that you can attach a remote debugger to. Once a connection is established, debug traffic is sent over HTTP to the remote application. You can use the spring.devtools.remote.debug.local-port property if you want to use a different port.

You’ll need to ensure that your remote application is started with remote debugging enabled. Often this can be achieved by configuring JAVA\_OPTS. For example, with Cloud Foundry you can add the following to your manifest.yml:

*---*

env:

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

|  |
| --- |
| [Tip] |
| Notice that you don’t need to pass an address=NNNN option to -Xrunjdwp. If omitted Java will simply pick a random free port. |
| [Note] |
| Debugging a remote service over the Internet can be slow and you might need to increase timeouts in your IDE. For example, in Eclipse you can select Java → Debug from Preferences…​ and change the Debugger timeout (ms) to a more suitable value (60000 works well in most situations). | |

|  |
| --- |
| [Warning] |
| When using the remote debug tunnel with IntelliJ IDEA, all breakpoints must be configured to suspend the thread rather than the VM. By default, breakpoints in IntelliJ IDEA suspend the entire VM rather than only suspending the thread that hit the breakpoint. This has the unwanted side-effect of suspending the thread that manages the remote debug tunnel, causing your debugging session to freeze. When using the remote debug tunnel with IntelliJ IDEA, all breakpoints should be configured to suspend the thread rather than the VM. Please see [IDEA-165769](https://youtrack.jetbrains.com/issue/IDEA-165769) for further details. |

**21. Packaging your application for production**

Executable jars can be used for production deployment. As they are self-contained, they are also ideally suited for cloud-based deployment.

For additional “production ready” features, such as health, auditing and metric REST or JMX end-points; consider adding spring-boot-actuator. See [*Part V, “Spring Boot Actuator: Production-ready features”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready) for details.

为生成打包你的应用：打包成可执行的jar包即可以用在生成环境，应为他们是自包含的，所以你也很好的支持基于云的部署。

**22. What to read next**

You should now have good understanding of how you can use Spring Boot along with some best practices that you should follow. You can now go on to learn about specific [*Spring Boot features*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features) in depth, or you could skip ahead and read about the “[production ready](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready)” aspects of Spring Boot.

**Part IV. Spring Boot features**

This section dives into the details of Spring Boot. Here you can learn about the key features that you will want to use and customize. If you haven’t already, you might want to read the [*Part II, “Getting started”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started) and [*Part III, “Using Spring Boot”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot) sections so that you have a good grounding of the basics.

**23. SpringApplication**

The SpringApplication class provides a convenient way to bootstrap a Spring application that will be started from a main() method. In many situations you can just delegate to the static SpringApplication.run method:

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

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

}

When your application starts you should see something similar to the following:

. \_\_\_\_ \_ \_\_ \_ \_

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

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

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

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

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

:: Spring Boot :: v1.5.7.RELEASE

2013-07-31 00:08:16.117 INFO 56603 --- [ main] o.s.b.s.app.SampleApplication : 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.AnnotationConfigEmbeddedWebApplication[[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection): startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy

2014-03-04 13:09:54.912 INFO 41370 --- [ main] .t.TomcatEmbeddedServletContainerFactory : Server initialized with port: 8080

2014-03-04 13:09:56.501 INFO 41370 --- [ main] o.s.b.s.app.SampleApplication : Started SampleApplication in 2.992 seconds (JVM running for 3.658)

By default INFO logging messages will be shown, including some relevant startup details such as the user that launched the application.

**23.1 Startup failure**

If your application fails to start, registered FailureAnalyzers get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance if you start a web application on port 8080 and that port is already in use, you should see something similar to the following:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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 provides numerous FailureAnalyzer implementations and you can [add your own](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-failure-analyzer) very easily. |

If no failure analyzers are able to handle the exception, you can still display the full auto-configuration report to better understand what went wrong. To do so you need to[enable the debug property](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config) or [enable DEBUG logging](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-log-levels) for org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer.

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

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

**23.2 Customizing the Banner**

The banner that is printed on start up can be changed by adding a banner.txt file to your classpath, or by setting banner.location to the location of such a file. If the file has an unusual encoding you can set banner.charset (default is UTF-8). In addition to a text file, you can also add a banner.gif, banner.jpg or banner.png image file to your classpath, or set a banner.image.location property. Images will be converted into an ASCII art representation and printed above any text banner.

Inside your banner.txt file you can use any of the following placeholders:

**Table 23.1. Banner variables**

| **Variable** | **Description** |
| --- | --- |
| ${application.version} | The version number of your application as declared in MANIFEST.MF. For exampleImplementation-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} | The Spring Boot version that you are using. For example 1.5.7.RELEASE. |
| ${spring-boot.formatted-version} | The Spring Boot version that you are using formatted for display (surrounded with brackets and prefixed with v). For example (v1.5.7.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/v1.5.7.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 exampleImplementation-Title: MyApp is printed as MyApp. |

|  |
| --- |
| [Tip] |
| The SpringApplication.setBanner(…​) method can be used if you want to generate a banner programmatically. Use the org.springframework.boot.Banner interface and implement your own printBanner() method. |

You can also use the spring.main.banner-mode property to determine if the banner has to be printed on System.out (console), using the configured logger (log) or not at all (off).

The printed banner will be registered as a singleton bean under the name springBootBanner.

|  |
| --- |
| [Note] |
| YAML maps off to false so make sure to add quotes if you want to disable the banner in your application.  spring:  main:  banner-mode: "off" |

**23.3 Customizing SpringApplication**

If the SpringApplication defaults aren’t to your taste you can instead create a local instance and customize it. For example, to turn off the banner you would write:

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

SpringApplication app = **new** SpringApplication(MySpringConfiguration.**class**);

app.setBannerMode(Banner.Mode.OFF);

app.run(args);

}

|  |
| --- |
| [Note] |
| The constructor arguments passed to SpringApplication are configuration sources for spring beans. In most cases these will be references to @Configuration classes, but they could also be references to XML configuration or to packages that should be scanned. |

It is also possible to configure the SpringApplication using an application.properties file. See [*Chapter 24, Externalized Configuration*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config) for details.

For a complete list of the configuration options, see the [SpringApplication Javadoc](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/SpringApplication.html).

**23.4 Fluent builder API**

If you need to build an ApplicationContext hierarchy (multiple contexts with a parent/child relationship), or if you just prefer using a ‘fluent’ builder API, you can use the SpringApplicationBuilder.

The SpringApplicationBuilder allows you to chain together multiple method calls, and includes parent and child methods that allow you to create a hierarchy.

For example:

**new** SpringApplicationBuilder()

.sources(Parent.**class**)

.child(Application.**class**)

.bannerMode(Banner.Mode.OFF)

.run(args);

|  |
| --- |
| [Note] |
| There are some restrictions when creating an ApplicationContext hierarchy, e.g. Web components **must** be contained within the child context, and the same Environment will be used for both parent and child contexts. See the [SpringApplicationBuilder Javadoc](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/builder/SpringApplicationBuilder.html) for full details. |

**23.5 Application events and listeners**

In addition to the usual Spring Framework events, such as [ContextRefreshedEvent](http://docs.spring.io/spring/docs/4.3.11.RELEASE/javadoc-api/org/springframework/context/event/ContextRefreshedEvent.html), a SpringApplication sends some additional application events.

|  |
| --- |
| [Note] |
| Some events are actually triggered before the ApplicationContext is created so you cannot register a listener on those as a @Bean. You can register them via the SpringApplication.addListeners(…​) or SpringApplicationBuilder.listeners(…​) methods.  If you want those listeners to be registered automatically regardless of the way the application is created you can add a META-INF/spring.factories file to your project and reference your listener(s) using the org.springframework.context.ApplicationListener key.  org.springframework.context.ApplicationListener=com.example.project.MyListener |

Application events are sent in the following order, as your application runs:

1. An ApplicationStartingEvent is sent at the start of a run, but before any processing except the registration of listeners and initializers.
2. An ApplicationEnvironmentPreparedEvent is sent when the Environment to be used in the context is known, but before the context is created.
3. An ApplicationPreparedEvent is sent just before the refresh is started, but after bean definitions have been loaded.
4. An ApplicationReadyEvent is sent after the refresh and any related callbacks have been processed to indicate the application is ready to service requests.
5. An ApplicationFailedEvent is sent if there is an exception on startup.

|  |
| --- |
| [Tip] |
| You often won’t need to use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks. |

**23.6 Web environment**

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

The algorithm used to determine a ‘web environment’ is fairly simplistic (based on the presence of a few classes). You can use setWebEnvironment(boolean webEnvironment) if you need to override the default.

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

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

**23.7 Accessing application arguments**

If you need to access the application arguments that were passed to SpringApplication.run(…​) you can inject aorg.springframework.boot.ApplicationArguments bean. The ApplicationArguments interface provides access to both the raw String[] arguments as well as parsed option and non-option arguments:

**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 will also register a CommandLinePropertySource with the Spring Environment. This allows you to also inject single application arguments using the @Value annotation. |

**23.8 Using the ApplicationRunner or CommandLineRunner**

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

The CommandLineRunner interfaces provides access to application arguments as a simple string array, whereas the ApplicationRunner uses the ApplicationArguments interface discussed above.

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

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

*@Component*

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

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

*// Do something...*

}

}

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

**23.9 Application exit**

Each SpringApplication will register a shutdown hook with the JVM to ensure that the ApplicationContext is closed gracefully on exit. All the standard Spring lifecycle callbacks (such as the DisposableBean interface, or the @PreDestroy annotation) can be used.

In addition, beans may implement the org.springframework.boot.ExitCodeGenerator interface if they wish to return a specific exit code when SpringApplication.exit() is called. This exit code can then be passed to System.exit() to return it as a status code.

*@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)));

}

}

Also, the ExitCodeGenerator interface may be implemented by exceptions. When such an exception is encountered, Spring Boot will return the exit code provided by the implemented getExitCode() method.

**23.10 Admin features**

It is possible to enable admin-related features for the application by specifying the spring.application.admin.enabled property. This exposes the[SpringApplicationAdminMXBean](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/admin/SpringApplicationAdminMXBean.java) on the platform MBeanServer. You could use this feature to administer your Spring Boot application remotely. This could also be useful for any service wrapper implementation.

|  |
| --- |
| [Tip] |
| If you want to know on which HTTP port the application is running, get the property with key local.server.port. |
| [Note] |
| Take care when enabling this feature as the MBean exposes a method to shutdown the application. | |

**24. Externalized Configuration**

Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables and command-line arguments to externalize configuration. Property values can be injected directly into your beans using the @Valueannotation, accessed via Spring’s Environment abstraction or [bound to structured objects](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-typesafe-configuration-properties) via @ConfigurationProperties.

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

1. [Devtools global settings properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-globalsettings) on your home directory (~/.spring-boot-devtools.properties when devtools is active).
2. [@TestPropertySource](http://docs.spring.io/spring/docs/4.3.11.RELEASE/javadoc-api/org/springframework/test/context/TestPropertySource.html) annotations on your tests.
3. [@SpringBootTest#properties](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/test/context/SpringBootTest.html) annotation attribute on your tests.
4. Command line arguments.
5. Properties from SPRING\_APPLICATION\_JSON (inline JSON embedded in an environment variable or system property)
6. ServletConfig init parameters.
7. ServletContext init parameters.
8. JNDI attributes from java:comp/env.
9. Java System properties (System.getProperties()).
10. OS environment variables.
11. A RandomValuePropertySource that only has properties in random.\*.
12. [Profile-specific application properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-profile-specific-properties) outside of your packaged jar (application-{profile}.properties and YAML variants)
13. [Profile-specific application properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-profile-specific-properties) packaged inside your jar (application-{profile}.properties and YAML variants)
14. Application properties outside of your packaged jar (application.properties and YAML variants).
15. Application properties packaged inside your jar (application.properties and YAML variants).
16. [@PropertySource](http://docs.spring.io/spring/docs/4.3.11.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotations on your @Configuration classes.
17. Default properties (specified using SpringApplication.setDefaultProperties).

To provide a concrete example, suppose you develop a @Component that uses a name property:

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

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

*@Component*

**public** **class** MyBean {

*@Value("${name}")*

**private** String name;

*// ...*

}

On your application classpath (e.g. inside your jar) you can have an application.properties that provides a sensible default property value for name. When running in a new environment, an application.properties can be provided outside of your jar that overrides the name; and for one-off testing, you can launch with a specific command line switch (e.g. java -jar app.jar --name="Spring").

|  |
| --- |
| [Tip] |
| The SPRING\_APPLICATION\_JSON properties can be supplied on the command line with an environment variable. For example in a UN\*X shell:  $ SPRING\_APPLICATION\_JSON='{"foo":{"bar":"spam"}}' java -jar myapp.jar  In this example you will end up with foo.bar=spam in the Spring Environment. You can also supply the JSON as spring.application.json in a System variable:  $ java -Dspring.application.json='{"foo":"bar"}' -jar myapp.jar  or command line argument:  $ java -jar myapp.jar --spring.application.json='{"foo":"bar"}'  or as a JNDI variable java:comp/env/spring.application.json. |

**24.1 Configuring random values**

The RandomValuePropertySource is useful for injecting random values (e.g. into secrets or test cases). It can produce integers, longs, uuids or strings, e.g.

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]}

The random.int\* syntax is OPEN value (,max) CLOSE where the OPEN,CLOSE are any character and value,max are integers. If max is provided then value is the minimum value and max is the maximum (exclusive).

**24.2 Accessing command line properties**

By default SpringApplication will convert any command line option arguments (starting with ‘--’, e.g. --server.port=9000) to a property and add it to the SpringEnvironment. As mentioned above, command line properties always take precedence over other property sources.

If you don’t want command line properties to be added to the Environment you can disable them using SpringApplication.setAddCommandLineProperties(false).

**24.3 Application property files**

SpringApplication will load properties from application.properties files in the following locations and add them to the Spring Environment:

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

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations).

|  |
| --- |
| [Note] |
| You can also [use YAML ('.yml') files](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-yaml) as an alternative to '.properties'. |

If you don’t like application.properties as the configuration file name you can switch to another by specifying a spring.config.name environment property. You can also refer to an explicit location using the spring.config.location environment property (comma-separated list of directory locations, or file paths).

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

or

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

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

If spring.config.location contains directories (as opposed to files) they should end in / (and will be appended with the names generated from spring.config.name before being loaded, including profile-specific file names). Files specified in spring.config.location are used as-is, with no support for profile-specific variants, and will be overridden by any profile-specific properties.

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

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

When custom config locations are configured, they are used in addition to the default locations. Custom locations are searched before the default locations. For example, if custom locations classpath:/custom-config/,file:./custom-config/ are configured, the search order becomes:

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

This search ordering allows you to specify default values in one configuration file and then selectively override those values in another. You can provide default values for you application in application.properties (or whatever other basename you choose with spring.config.name) in one of the default locations. These default values can then be overriden at runtime with a different file located in one of the custom locations.

|  |
| --- |
| [Note] |
| If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (e.g. SPRING\_CONFIG\_NAME instead of spring.config.name). |
| [Note] |
| If you are running in a container then JNDI properties (in java:comp/env) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties. | |

**24.4 Profile-specific properties**

In addition to application.properties files, profile-specific properties can also be defined using the naming convention application-{profile}.properties. TheEnvironment has a set of default profiles (by default [default]) which are used if no active profiles are set (i.e. if no profiles are explicitly activated then properties from application-default.properties are loaded).

Profile-specific properties are loaded from the same locations as standard application.properties, with profile-specific files always overriding the non-specific ones irrespective of whether the profile-specific files are inside or outside your packaged jar.

If several profiles are specified, a last wins strategy applies. For example, profiles specified by the spring.profiles.active property are added after those configured via the SpringApplication API and therefore take precedence.

|  |
| --- |
| [Note] |
| If you have specified any files in spring.config.location, profile-specific variants of those files will not be considered. Use directories inspring.config.location if you also want to also use profile-specific properties. |

**24.5 Placeholders in properties**

The values in application.properties are filtered through the existing Environment when they are used so you can refer back to previously defined values (e.g. from System properties).

app.name=MyApp

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

|  |
| --- |
| [Tip] |
| You can also use this technique to create ‘short’ variants of existing Spring Boot properties. See the [*Section 72.4, “Use ‘short’ command line arguments”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-short-command-line-arguments)how-to for details. |

**24.6 Using YAML instead of Properties**

[YAML](http://yaml.org/) is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data. The SpringApplication class will automatically support YAML as an alternative to properties whenever you have the [SnakeYAML](http://www.snakeyaml.org/) library on your classpath.

|  |
| --- |
| [Note] |
| If you use ‘Starters’ SnakeYAML will be automatically provided via spring-boot-starter. |

**24.6.1 Loading YAML**

Spring Framework provides two convenient classes that can be used to load YAML documents. The YamlPropertiesFactoryBean will load YAML as Properties and the YamlMapFactoryBean will load YAML as a Map.

For example, the following YAML document:

environments:

dev:

url: http://dev.bar.com

name: Developer Setup

prod:

url: http://foo.bar.com

name: My Cool App

Would be transformed into these properties:

environments.dev.url=http://dev.bar.com

environments.dev.name=Developer Setup

environments.prod.url=http://foo.bar.com

environments.prod.name=My Cool App

YAML lists are represented as property keys with [index] dereferencers, for example this YAML:

my:

servers:

- dev.bar.com

- foo.bar.com

Would be transformed into these properties:

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

my.servers[1]=foo.bar.com

To bind to properties like that using the Spring DataBinder utilities (which is what @ConfigurationProperties does) you need to have a property in the target bean of type java.util.List (or Set) and you either need to provide a setter, or initialize it with a mutable value, e.g. this will bind to the properties above

*@ConfigurationProperties(prefix="my")*

**public** **class** Config {

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

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

**return** **this**.servers;

}

}

|  |
| --- |
| [Note] |
| Extra care is required when configuring lists that way as overriding will not work as you would expect. In the example above, when my.servers is redefined in several places, the individual elements are targeted for override, not the list. To make sure that a PropertySource with higher precedence can override the list, you need to define it as a single property:  my:  servers: dev.bar.com,foo.bar.com |

**24.6.2 Exposing YAML as properties in the Spring Environment**

The YamlPropertySourceLoader class can be used to expose YAML as a PropertySource in the Spring Environment. This allows you to use the familiar @Valueannotation with placeholders syntax to access YAML properties.

**24.6.3 Multi-profile YAML documents**

You can specify multiple profile-specific YAML documents in a single file by using a spring.profiles key to indicate when the document applies. For example:

server:

address: 192.168.1.100

*---*

spring:

profiles: development

server:

address: 127.0.0.1

*---*

spring:

profiles: production

server:

address: 192.168.1.120

In the example above, the server.address property will be 127.0.0.1 if the development profile is active. If the development and production profiles are **not**enabled, then the value for the property will be 192.168.1.100.

The default profiles are activated if none are explicitly active when the application context starts. So in this YAML we set a value for security.user.password that is**only** available in the "default" profile:

server:

port: 8000

*---*

spring:

profiles: default

security:

user:

password: weak

whereas in this example, the password is always set because it isn’t attached to any profile, and it would have to be explicitly reset in all other profiles as necessary:

server:

port: 8000

security:

user:

password: weak

Spring profiles designated using the "spring.profiles" element may optionally be negated using the ! character. If both negated and non-negated profiles are specified for a single document, at least one non-negated profile must match and no negated profiles may match.

**24.6.4 YAML shortcomings**

YAML files can’t be loaded via the @PropertySource annotation. So in the case that you need to load values that way, you need to use a properties file.

**24.6.5 Merging YAML lists**

As [we have seen above](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-loading-yaml), any YAML content is ultimately transformed to properties. That process may be counter intuitive when overriding “list” properties via a profile.

For example, assume a MyPojo object with name and description attributes that are null by default. Let’s expose a list of MyPojo from FooProperties:

*@ConfigurationProperties("foo")*

**public** **class** FooProperties {

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

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

**return** **this**.list;

}

}

Consider the following configuration:

foo:

list:

- name: my name

description: my description

*---*

spring:

profiles: dev

foo:

list:

- name: my another name

If the dev profile isn’t active, FooProperties.list will contain one MyPojo entry as defined above. If the dev profile is enabled however, the list will *still* only contain one entry (with name “my another name” and description null). This configuration *will not* add a second MyPojo instance to the list, and it won’t merge the items.

When a collection is specified in multiple profiles, the one with highest priority is used (and only that one):

foo:

list:

- name: my name

description: my description

- name: another name

description: another description

*---*

spring:

profiles: dev

foo:

list:

- name: my another name

In the example above, considering that the dev profile is active, FooProperties.list will contain *one* MyPojo entry (with name “my another name” and description null).

**24.7 Type-safe Configuration Properties**

Using the @Value("${property}") annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or your data is hierarchical in nature. Spring Boot provides an alternative method of working with properties that allows strongly typed beans to govern and validate the configuration of your application.

**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) { ... }

}

}

The POJO above defines the following properties:

* foo.enabled, false by default
* foo.remote-address, with a type that can be coerced from String
* foo.security.username, with a nested "security" whose name is determined by the name of the property. In particular the return type is not used at all there and could have been SecurityProperties
* foo.security.password
* foo.security.roles, with a collection of String

|  |
| --- |
| [Note] |
| Getters and setters are usually mandatory, since binding is via standard Java Beans property descriptors, just like in Spring MVC. There are cases where a setter may be omitted:   * Maps, as long as they are initialized, need a getter but not necessarily a setter since they can be mutated by the binder. * Collections and arrays can be either accessed via an index (typically with YAML) or using a single comma-separated value (properties). In the latter case, a setter is mandatory. We recommend to always add a setter for such types. If you initialize a collection, make sure it is not immutable (as in the example above) * If nested POJO properties are initialized (like the Security field in the example above), a setter is not required. If you want the binder to create the instance on-the-fly using its default constructor, you will need a setter.   Some people use Project Lombok to add getters and setters automatically. Make sure that Lombok doesn’t generate any particular constructor for such type as it will be used automatically by the container to instantiate the object. |
| [Tip] |
| See also the [differences between @Value and @ConfigurationProperties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-vs-value). | |

You also need to list the properties classes to register in the @EnableConfigurationProperties annotation:

*@Configuration*

*@EnableConfigurationProperties(FooProperties.class)*

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

}

|  |
| --- |
| [Note] |
| When @ConfigurationProperties bean is registered that way, the bean will have a conventional name: <prefix>-<fqn>, where <prefix> is the environment key prefix specified in the @ConfigurationProperties annotation and <fqn> the fully qualified name of the bean. If the annotation does not provide any prefix, only the fully qualified name of the bean is used.  The bean name in the example above will be foo-com.example.FooProperties. |

Even if the configuration above will create a regular bean for FooProperties, we recommend that @ConfigurationProperties only deal with the environment and in particular does not inject other beans from the context. Having said that, The @EnableConfigurationProperties annotation is *also* automatically applied to your project so that any *existing* bean annotated with @ConfigurationProperties will be configured from the Environment. You could shortcut MyConfiguration above by making sure FooProperties is a already a bean:

*@Component*

*@ConfigurationProperties(prefix="foo")*

**public** **class** FooProperties {

*// ... see above*

}

This style of configuration works particularly well with the SpringApplication external YAML configuration:

*# application.yml*

foo:

remote-address: 192.168.1.1

security:

username: foo

roles:

- USER

- ADMIN

*# additional configuration as required*

To work with @ConfigurationProperties beans you can just inject them in the same way as any other bean.

*@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] |
| Using @ConfigurationProperties also allows you to generate meta-data files that can be used by IDEs to offer auto-completion for your own keys, see the [Appendix B, *Configuration meta-data*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata) appendix for details. |

**24.7.1 Third-party configuration**

As well as using @ConfigurationProperties to annotate a class, you can also use it on public @Bean methods. This can be particularly useful when you want to bind properties to third-party components that are outside of your control.

To configure a bean from the Environment properties, add @ConfigurationProperties to its bean registration:

*@ConfigurationProperties(prefix = "bar")*

*@Bean*

**public** BarComponent barComponent() {

...

}

Any property defined with the bar prefix will be mapped onto that BarComponent bean in a similar manner as the FooProperties example above.

**24.7.2 Relaxed binding**

Spring Boot uses some relaxed rules for binding Environment properties to @ConfigurationProperties beans, so there doesn’t need to be an exact match between the Environment property name and the bean property name. Common examples where this is useful include dashed separated (e.g. context-path binds to contextPath), and capitalized (e.g. PORT binds to port) environment properties.

For example, given the following @ConfigurationProperties class:

*@ConfigurationProperties(prefix="person")*

**public** **class** OwnerProperties {

**private** String firstName;

**public** String getFirstName() {

**return** **this**.firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

}

The following properties names can all be used:

**Table 24.1. relaxed binding**

| **Property** | **Note** |
| --- | --- |
| person.firstName | Standard camel case syntax. |
| person.first-name | Dashed notation, recommended for use in .properties and .yml files. |
| person.first\_name | Underscore notation, alternative format for use in .properties and .yml files. |
| PERSON\_FIRST\_NAME | Upper case format. Recommended when using a system environment variables. |

**24.7.3 Properties conversion**

Spring will attempt to coerce the external application properties to the right type when it binds to the @ConfigurationProperties beans. If you need custom type conversion you can provide a ConversionService bean (with bean id conversionService) or custom property editors (via a CustomEditorConfigurer bean) or custom Converters (with bean definitions annotated as @ConfigurationPropertiesBinding).

|  |
| --- |
| [Note] |
| As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your ConversionService is using. Typically, any dependency that you require may not be fully initialized at creation time. You may want to rename your custom ConversionService if it’s not required for configuration keys coercion and only rely on custom converters qualified with @ConfigurationPropertiesBinding. |

**24.7.4 @ConfigurationProperties Validation**

Spring Boot will attempt to validate @ConfigurationProperties classes whenever they are annotated with Spring’s @Validated annotation. You can use JSR-303 javax.validation constraint annotations directly on your configuration class. Simply ensure that a compliant JSR-303 implementation is on your classpath, then add constraint annotations to your fields:

*@ConfigurationProperties(prefix="foo")*

*@Validated*

**public** **class** FooProperties {

*@NotNull*

**private** InetAddress remoteAddress;

*// ... getters and setters*

}

In order to validate values of nested properties, you must annotate the associated field as @Valid to trigger its validation. For example, building upon the aboveFooProperties example:

*@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*

}

}

You can also add a custom Spring Validator by creating a bean definition called configurationPropertiesValidator. The @Bean method should be declared static. The configuration properties validator is created very early in the application’s lifecycle and declaring the @Bean method as static allows the bean to be created without having to instantiate the @Configuration class. This avoids any problems that may be caused by early instantiation. There is a [property validation sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-property-validation) so you can see how to set things up.

|  |
| --- |
| [Tip] |
| The spring-boot-actuator module includes an endpoint that exposes all @ConfigurationProperties beans. Simply point your web browser to /configprops or use the equivalent JMX endpoint. See the [*Production ready features*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints). section for details. |

**24.7.5 @ConfigurationProperties vs. @Value**

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

| **Feature** | **@ConfigurationProperties** | **@Value** |
| --- | --- | --- |
| [Relaxed binding](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-relaxed-binding) | Yes | No |
| [Meta-data support](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata) | Yes | No |
| SpEL evaluation | No | Yes |

If you define a set of configuration keys for your own components, we recommend you to group them in a POJO annotated with @ConfigurationProperties. Please also be aware that since @Value does not support relaxed binding, it isn’t a great candidate if you need to provide the value using environment variables.

Finally, while you can write a SpEL expression in @Value, such expressions are not processed from [Application property files](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-application-property-files).

**25. Profiles**

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. Any @Component or @Configuration can be marked with @Profile to limit when it is loaded:

*@Configuration*

*@Profile("production")*

**public** **class** ProductionConfiguration {

*// ...*

}

In the normal Spring way, you can use a spring.profiles.active Environment property to specify which profiles are active. You can specify the property in any of the usual ways, for example you could include it in your application.properties:

spring.profiles.active=dev,hsqldb

or specify on the command line using the switch --spring.profiles.active=dev,hsqldb.

**25.1 Adding active profiles**

The spring.profiles.active property follows the same ordering rules as other properties, the highest PropertySource will win. This means that you can specify active profiles in application.properties then **replace** them using the command line switch.

Sometimes it is useful to have profile-specific properties that **add** to the active profiles rather than replace them. The spring.profiles.include property can be used to unconditionally add active profiles. The SpringApplication entry point also has a Java API for setting additional profiles (i.e. on top of those activated by thespring.profiles.active property): see the setAdditionalProfiles() method.

For example, when an application with following properties is run using the switch --spring.profiles.active=prod the proddb and prodmq profiles will also be activated:

*---*

my.property: fromyamlfile

*---*

spring.profiles: prod

spring.profiles.include:

- proddb

- prodmq

|  |
| --- |
| [Note] |
| Remember that the spring.profiles property can be defined in a YAML document to determine when this particular document is included in the configuration. See [Section 72.7, “Change configuration depending on the environment”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-change-configuration-depending-on-the-environment) for more details. |

**25.2 Programmatically setting profiles**

You can programmatically set active profiles by calling SpringApplication.setAdditionalProfiles(…​) before your application runs. It is also possible to activate profiles using Spring’s ConfigurableEnvironment interface.

**25.3 Profile-specific configuration files**

Profile-specific variants of both application.properties (or application.yml) and files referenced via @ConfigurationProperties are considered as files are loaded. See [*Section 24.4, “Profile-specific properties”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-profile-specific-properties) for details.

**26. Logging**

Spring Boot uses [Commons Logging](https://commons.apache.org/logging) for all internal logging, but leaves the underlying log implementation open. Default configurations are provided for [Java Util Logging](https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html),[Log4J2](https://logging.apache.org/log4j/2.x/) and [Logback](http://logback.qos.ch/). In each case loggers are pre-configured to use console output with optional file output also available.

By default, If you use the ‘Starters’, Logback will be used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J or SLF4J will all work correctly.

|  |
| --- |
| [Tip] |
| There are a lot of logging frameworks available for Java. Don’t worry if the above list seems confusing. Generally you won’t need to change your logging dependencies and the Spring Boot defaults will work just fine. |

**26.1 Log format**

The default log output from Spring Boot looks like this:

2014-03-05 10:57:51.112 INFO 45469 --- [ main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/7.0.52

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: [/\*]

The following items are output:

* Date and Time — Millisecond precision and easily sortable.
* Log Level — ERROR, WARN, INFO, DEBUG or TRACE.
* Process ID.
* A --- separator to distinguish the start of actual log messages.
* Thread name — Enclosed in square brackets (may be truncated for console output).
* Logger name — This is usually the source class name (often abbreviated).
* The log message.

|  |
| --- |
| [Note] |
| Logback does not have a FATAL level (it is mapped to ERROR) |

**26.2 Console output**

The default log configuration will echo messages to the console as they are written. By default ERROR, WARN and INFO level messages are logged. You can also enable a “debug” mode by starting your application with a --debug flag.

$ java -jar myapp.jar --debug

|  |
| --- |
| [Note] |
| you can also specify debug=true in your application.properties. |

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with DEBUG level.

Alternatively, you can enable a “trace” mode by starting your application with a --trace flag (or trace=true in your application.properties). This will enable trace logging for a selection of core loggers (embedded container, Hibernate schema generation and the whole Spring portfolio).

**26.2.1 Color-coded output**

If your terminal supports ANSI, color output will be used to aid readability. You can set spring.output.ansi.enabled to a [supported value](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/ansi/AnsiOutput.Enabled.html) to override the auto detection.

Color coding is configured using the %clr conversion word. In its simplest form the converter will color the output according to the log level, for example:

%clr(%5p)

The mapping of log level to a color is as follows:

| **Level** | **Color** |
| --- | --- |
| FATAL | Red |
| ERROR | Red |
| WARN | Yellow |
| INFO | Green |
| DEBUG | Green |
| TRACE | Green |

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow:

%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}

The following colors and styles are supported:

* blue
* cyan
* faint
* green
* magenta
* red
* yellow

**26.3 File output**

By default, Spring Boot will only log to the console and will not write log files. If you want to write log files in addition to the console output you need to set alogging.file or logging.path property (for example in your application.properties).

The following table shows how the logging.\* properties can be used together:

**Table 26.1. Logging properties**

| **logging.file** | **logging.path** | **Example** | **Description** |
| --- | --- | --- | --- |
| *(none)* | *(none)* |  | 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. |

Log files will rotate when they reach 10 MB and as with console output, ERROR, WARN and INFO level messages are logged by default.

|  |
| --- |
| [Note] |
| The logging system is initialized early in the application lifecycle and as such logging properties will not be found in property files loaded via @PropertySource annotations. |
| [Tip] |
| Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as logback.configurationFile for Logback) are not managed by spring Boot. | |

**26.4 Log Levels**

All the supported logging systems can have the logger levels set in the Spring Environment (so for example in application.properties) using ‘logging.level.\*=LEVEL’ where ‘LEVEL’ is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF. The root logger can be configured using logging.level.root. Example application.properties:

logging.level.root=WARN

logging.level.org.springframework.web=DEBUG

logging.level.org.hibernate=ERROR

|  |
| --- |
| [Note] |
| By default Spring Boot remaps Thymeleaf INFO messages so that they are logged at DEBUG level. This helps to reduce noise in the standard log output. See [LevelRemappingAppender](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/logging/logback/LevelRemappingAppender.java) for details of how you can apply remapping in your own configuration. |

**26.5 Custom log configuration**

The various logging systems can be activated by including the appropriate libraries on the classpath, and further customized by providing a suitable configuration file in the root of the classpath, or in a location specified by the Spring Environment property logging.config.

You can force Spring Boot to use a particular logging system using the org.springframework.boot.logging.LoggingSystem system property. The value should be the fully-qualified class name of a LoggingSystem implementation. You can also disable Spring Boot’s logging configuration entirely by using a value of none.

|  |
| --- |
| [Note] |
| Since logging is initialized **before** the ApplicationContext is created, it isn’t possible to control logging from @PropertySources in Spring @Configuration files. System properties and the conventional Spring Boot external configuration files work just fine.) |

Depending on your logging system, the following files will be loaded:

| **Logging System** | **Customization** |
| --- | --- |
| Logback | logback-spring.xml, logback-spring.groovy, logback.xml or logback.groovy |
| Log4j2 | log4j2-spring.xml or log4j2.xml |
| JDK (Java Util Logging) | logging.properties |
| [Note] |
| When possible we recommend that you use the -spring variants for your logging configuration (for example logback-spring.xml rather than logback.xml). If you use standard configuration locations, Spring cannot completely control log initialization. |

|  |
| --- |
| [Warning] |
| There are known classloading issues with Java Util Logging that cause problems when running from an ‘executable jar’. We recommend that you avoid it if at all possible. |

To help with the customization some other properties are transferred from the Spring Environment to System properties:

| **Spring Environment** | **System Property** | **Comments** |
| --- | --- | --- |
| logging.exception-conversion-word | LOG\_EXCEPTION\_CONVERSION\_WORD | The conversion word that’s used when logging exceptions. |
| logging.file | LOG\_FILE | Used in default log configuration if defined. |
| logging.path | LOG\_PATH | Used in default log configuration if defined. |
| logging.pattern.console | CONSOLE\_LOG\_PATTERN | The log pattern to use on the console (stdout). (Only supported with the default logback setup.) |
| logging.pattern.file | FILE\_LOG\_PATTERN | The log pattern to use in a file (if LOG\_FILE enabled). (Only supported with the default logback setup.) |
| logging.pattern.level | LOG\_LEVEL\_PATTERN | 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). |

All the logging systems supported can consult System properties when parsing their configuration files. See the default configurations in spring-boot.jar for examples.

|  |
| --- |
| [Tip] |
| If you want to use a placeholder in a logging property, you should use [Spring Boot’s syntax](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-placeholders-in-properties) and not the syntax of the underlying framework. Notably, if you’re using Logback, you should use : as the delimiter between a property name and its default value and not :-. |
| [Tip] |
| You can add MDC and other ad-hoc content to log lines by overriding only the LOG\_LEVEL\_PATTERN (or logging.pattern.level with Logback). For example, if you use logging.pattern.level=user:%X{user} %5p then the default log format will contain an MDC entry for "user" if it exists, e.g.  2015-09-30 12:30:04.031 user:juergen INFO 22174 --- [ nio-8080-exec-0] demo.Controller  Handling authenticated request | |

**26.6 Logback extensions**

Spring Boot includes a number of extensions to Logback which can help with advanced configuration. You can use these extensions in your logback-spring.xmlconfiguration file.

|  |
| --- |
| [Note] |
| You cannot use extensions in the standard logback.xml configuration file since it’s loaded too early. You need to either use logback-spring.xml or define a logging.config property. |
| [Warning] |
| The extensions cannot be used with Logback’s [configuration scanning](http://logback.qos.ch/manual/configuration.html#autoScan). If you attempt to do so, making changes to the configuration file will result in an error similar to one of the following being logged: | |

ERROR in [[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection):71 - no applicable action for [springProperty], current ElementPath is [[configuration][springProperty]]

ERROR in [[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection):71 - no applicable action for [springProfile], current ElementPath is [[configuration][springProfile]]

**26.6.1 Profile-specific configuration**

The <springProfile> tag allows you to optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the <configuration> element. Use the name attribute to specify which profile accepts the configuration. Multiple profiles can be specified using a comma-separated list.

<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>

**26.6.2 Environment properties**

The <springProperty> tag allows you to surface properties from the Spring Environment for use within Logback. This can be useful if you want to access values from your application.properties file in your logback configuration. The tag works in a similar way to Logback’s standard <property> tag, but rather than specifying a direct value you specify the source of the property (from the Environment). You can use the scope attribute if you need to store the property somewhere other than in local scope. If you need a fallback value in case the property is not set in the Environment, you can use the defaultValue attribute.

<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] |
| The RelaxedPropertyResolver is used to access Environment properties. If specify the source in dashed notation (my-property-name) all the relaxed variations will be tried (myPropertyName, MY\_PROPERTY\_NAME etc). |

**27. Developing web applications**

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the spring-boot-starter-web module to get up and running quickly.

If you haven’t yet developed a Spring Boot web application you can follow the "Hello World!" example in the [*Getting started*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application) section.

**27.1 The ‘Spring Web MVC framework’**

The Spring Web MVC framework (often referred to as simply ‘Spring MVC’) is a rich ‘model view controller’ web framework. Spring MVC lets you create special @Controller or @RestController beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP using @RequestMapping annotations.

Here is a typical example @RestController to serve JSON data:

*@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 is part of the core Spring Framework and detailed information is available in the [reference documentation](http://docs.spring.io/spring/docs/4.3.11.RELEASE/spring-framework-reference/htmlsingle#mvc). There are also several guides available at [spring.io/guides](https://spring.io/guides) that cover Spring MVC.

**27.1.1 Spring MVC auto-configuration**

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring’s defaults:

* Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.
* Support for serving static resources, including support for WebJars (see below).
* Automatic registration of Converter, GenericConverter, Formatter beans.
* Support for HttpMessageConverters (see below).
* Automatic registration of MessageCodesResolver (see below).
* Static index.html support.
* Custom Favicon support (see below).
* Automatic use of a ConfigurableWebBindingInitializer bean (see below).

If you want to keep Spring Boot MVC features, and you just want to add additional [MVC configuration](http://docs.spring.io/spring/docs/4.3.11.RELEASE/spring-framework-reference/htmlsingle#mvc) (interceptors, formatters, view controllers etc.) you can add your own @Configuration class of type WebMvcConfigurerAdapter, but **without** @EnableWebMvc. If you wish to provide custom instances of RequestMappingHandlerMapping, RequestMappingHandlerAdapter or ExceptionHandlerExceptionResolver you can declare a WebMvcRegistrationsAdapter instance providing such components.

If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc.

**27.1.2 HttpMessageConverters**

Spring MVC uses the HttpMessageConverter interface to convert HTTP requests and responses. Sensible defaults are included out of the box, for example Objects can be automatically converted to JSON (using the Jackson library) or XML (using the Jackson XML extension if available, else using JAXB). Strings are encoded using UTF-8 by default.

If you need to add or customize converters you can use Spring Boot’s HttpMessageConverters class:

**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);

}

}

Any HttpMessageConverter bean that is present in the context will be added to the list of converters. You can also override default converters that way.

**27.1.3 Custom JSON Serializers and Deserializers**

If you’re using Jackson to serialize and deserialize JSON data, you might want to write your own JsonSerializer and JsonDeserializer classes. Custom serializers are usually [registered with Jackson via a Module](http://wiki.fasterxml.com/JacksonHowToCustomDeserializers), but Spring Boot provides an alternative @JsonComponent annotation which makes it easier to directly register Spring Beans.

You can use @JsonComponent directly on JsonSerializer or JsonDeserializer implementations. You can also use it on classes that contains serializers/deserializers as inner-classes. For example:

**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> {

*// ...*

}

}

All @JsonComponent beans in the ApplicationContext will be automatically registered with Jackson, and since @JsonComponent is meta-annotated with @Component, the usual component-scanning rules apply.

Spring Boot also provides [JsonObjectSerializer](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jackson/JsonObjectSerializer.java) and [JsonObjectDeserializer](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jackson/JsonObjectDeserializer.java) base classes which provide useful alternatives to the standard Jackson versions when serializing Objects. See the Javadoc for details.

**27.1.4 MessageCodesResolver**

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: MessageCodesResolver. Spring Boot will create one for you if you set the spring.mvc.message-codes-resolver.format property PREFIX\_ERROR\_CODE or POSTFIX\_ERROR\_CODE (see the enumeration in DefaultMessageCodesResolver.Format).

**27.1.5 Static Content**

By default Spring Boot will serve static content from a directory called /static (or /public or /resources or /META-INF/resources) in the classpath or from the root of the ServletContext. It uses the ResourceHttpRequestHandler from Spring MVC so you can modify that behavior by adding your own WebMvcConfigurerAdapter and overriding the addResourceHandlers method.

In a stand-alone web application the default servlet from the container is also enabled, and acts as a fallback, serving content from the root of the ServletContext if Spring decides not to handle it. Most of the time this will not happen (unless you modify the default MVC configuration) because Spring will always be able to handle requests through the DispatcherServlet.

By default, resources are mapped on /\*\* but you can tune that via spring.mvc.static-path-pattern. For instance, relocating all resources to /resources/\*\*can be achieved as follows:

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

You can also customize the static resource locations using spring.resources.static-locations (replacing the default values with a list of directory locations). If you do this the default welcome page detection will switch to your custom locations, so if there is an index.html in any of your locations on startup, it will be the home page of the application.

In addition to the ‘standard’ static resource locations above, a special case is made for [Webjars content](http://www.webjars.org/). Any resources with a path in /webjars/\*\* will be served from jar files if they are packaged in the Webjars format.

|  |
| --- |
| [Tip] |
| Do not use the src/main/webapp directory if your application will be packaged as a jar. Although this directory is a common standard, it will **only** work with war packaging and it will be silently ignored by most build tools if you generate a jar. |

Spring Boot also supports advanced resource handling features provided by Spring MVC, allowing use cases such as cache busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, simply add the webjars-locator dependency. Then declare your Webjar, taking jQuery for example, as "/webjars/jquery/dist/jquery.min.js" which results in "/webjars/jquery/x.y.z/dist/jquery.min.js" where x.y.z is the Webjar version.

|  |
| --- |
| [Note] |
| If you are using JBoss, you’ll need to declare the webjars-locator-jboss-vfs dependency instead of the webjars-locator; otherwise all Webjars resolve as a 404. |

To use cache busting, the following configuration will configure a cache busting solution for all static resources, effectively adding a content hash in URLs, such as<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>:

spring.resources.chain.strategy.content.enabled=true

spring.resources.chain.strategy.content.paths=/\*\*

|  |
| --- |
| [Note] |
| Links to resources are rewritten at runtime in template, thanks to a ResourceUrlEncodingFilter, auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines aren’t automatically supported right now, but can be with custom template macros/helpers and the use of the [ResourceUrlProvider](http://docs.spring.io/spring/docs/4.3.11.RELEASE/javadoc-api/org/springframework/web/servlet/resource/ResourceUrlProvider.html). |

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That’s why other strategies are also supported and can be combined. A "fixed" strategy will add a static version string in the URL, without changing the file name:

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

With this configuration, JavaScript modules located under "/js/lib/" will use a fixed versioning strategy "/v12/js/lib/mymodule.js" while other resources will still use the content one <link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>.

See [ResourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ResourceProperties.java) for more of the supported options.

|  |
| --- |
| [Tip] |
| This feature has been thoroughly described in a dedicated [blog post](https://spring.io/blog/2014/07/24/spring-framework-4-1-handling-static-web-resources) and in Spring Framework’s [reference documentation](http://docs.spring.io/spring/docs/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#mvc-config-static-resources). |

**27.1.6 Custom Favicon**

Spring Boot looks for a favicon.ico in the configured static content locations and the root of the classpath (in that order). If such file is present, it is automatically used as the favicon of the application.

**27.1.7 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.

**27.1.8 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](https://mustache.github.io/)

|  |
| --- |
| [Tip] |
| JSPs should be avoided if possible, there are several [known limitations](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jsp-limitations) 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/. |

**27.1.9 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 implementErrorController 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 CustomerErrorTypePOJO 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#mvc-exceptionhandlers) and [@ControllerAdvice](http://docs.spring.io/spring/docs/4.3.11.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

**27.1.10 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.

**27.1.11 CORS support**

[Cross-origin resource sharing](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing) (CORS) is a [W3C specification](https://www.w3.org/TR/cors/) implemented by [most browsers](http://caniuse.com/#feat=cors) 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#cors) out of the box. Using [controller method CORS configuration](http://docs.spring.io/spring/docs/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#_controller_method_cors_configuration) with [@CrossOrigin](http://docs.spring.io/spring/docs/4.3.11.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/4.3.11.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/\*\*");

}

};

}

}

**27.2 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/classeswhen 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 isjerseyFilterRegistration). 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/v1.5.7.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/v1.5.7.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.

**27.3 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. |

**27.3.1 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.

**27.3.2 Servlet Context Initialization**

Embedded servlet containers will not directly execute the Servlet 3.0+ javax.servlet.ServletContainerInitializer interface, or Spring’sorg.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 theorg.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. |

**27.3.3 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 ApplicationContextand EmbeddedServletContainerFactory will be created on your behalf. |

**27.3.4 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-ssl)
* [HTTP compression](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#how-to-enable-http-response-compression)

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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-accesslogs) can be configured with specific features of the embedded servlet container.

|  |
| --- |
| [Tip] |
| See the [ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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 ConfigurableEmbeddedServletContainerwhich 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, JettyEmbeddedServletContainerFactoryor 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.

**27.3.5 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling), [custom error pages](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling-custom-error-pages) should be used instead.

There is a [JSP sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-web-jsp) so you can see how to set things up.

**28. 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 INFOmessages, 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/v1.5.7.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 (SpringBootWebSecurityConfigurationfor 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/v1.5.7.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. |

**28.1 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 Security OAuth 2 Developers Guide](http://projects.spring.io/spring-security-oauth/docs/oauth2.html).

**28.1.1 Authorization Server**

To create an Authorization Server and grant access tokens you need to use @EnableAuthorizationServer and provide security.oauth2.client.client-id andsecurity.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:[[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection):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.

**28.1.2 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).

**28.2 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.

**28.3 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----- |

**28.3.1 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). | |

**28.3.2 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();

}

}

**28.4 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.

**29. 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.

**29.1 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-a-datasource) for more advanced examples, typically to take full control over the configuration of the DataSource. |

**29.1.1 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*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-database-initialization) |

Spring Boot can auto-configure embedded [H2](http://www.h2database.com/), [HSQL](http://hsqldb.org/) and [Derby](https://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 viaspring-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. | |

**29.1.2 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 inapplication.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/v1.5.7.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](https://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

**29.1.3 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.propertiesshows how you can access a JBoss AS defined DataSource:

spring.datasource.jndi-name=java:jboss/datasources/customers

**29.2 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;

}

*// ...*

}

**29.3 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 Data with JPA’](https://spring.io/guides/gs/accessing-data-jpa/) guide from [spring.io](https://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/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-hibernate4) and [Hibernate 5.2](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-hibernate52)samples to see how to do so. | |

**29.3.1 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”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-separate-entity-definitions-from-spring-configuration) how-to. |

**29.3.2 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/). |

**29.3.3 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 usingspring.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.

**29.3.4 Open EntityManager in View**

If you are running a web application, Spring Boot will by default register [OpenEntityManagerInViewInterceptor](http://docs.spring.io/spring/docs/4.3.11.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.

**29.4 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools)

|  |
| --- |
| [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. |

**29.4.1 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.

**29.4.2 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

**29.5 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.

**29.5.1 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>

**29.5.2 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);

}

**29.5.3 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.

**30. 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).

**30.1 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.

**30.1.1 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.

**30.2 MongoDB**

[MongoDB](https://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’.

**30.2.1 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:[[email protected]](https://docs.spring.io/cdn-cgi/l/email-protection):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.

**30.2.2 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.

**30.2.3 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/). |

**30.2.4 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 onde.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.

**30.3 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’.

**30.3.1 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 typeNeo4jOperations 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

**30.3.2 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:getting-started) for more details. |

**30.3.3 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

**30.3.4 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

**30.3.5 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/). |

**30.4 Gemfire**

[Spring Data Gemfire](https://github.com/spring-projects/spring-data-gemfire) provides convenient Spring-friendly tools for accessing the [Pivotal Gemfire](https://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).

**30.5 Solr**

[Apache Solr](https://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.

**30.5.1 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](http://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.

**30.5.2 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/). |

**30.6 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 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).

**30.6.1 Connecting to Elasticsearch using Jest**

If you have Jest on the classpath, you can inject an auto-configured JestClient targeting [localhost:9200](http://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.

**30.6.2 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.

**30.6.3 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 thatCity 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/). |

**30.7 Cassandra**

[Cassandra](https://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.

**30.7.1 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.

**30.7.2 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/). |

**30.8 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.

**30.8.1 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.

**30.8.2 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. |

**30.9 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.

**30.9.1 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.

**30.9.2 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;

}

*// ...*

}

**30.9.3 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.enabledproperty. If you have custom attributes, you can use spring.ldap.embedded.validation.schema to define your custom attribute types or object classes.

**31. 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/4.3.11.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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-simple) 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#cache-annotations-put) or [evict](http://docs.spring.io/spring/docs/4.3.11.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. | |

**31.1 Supported cache providers**

The cache abstraction does not provide an actual store and relies on abstraction materialized by the org.springframework.cache.Cache andorg.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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-generic)
* [JCache (JSR-107)](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-jcache) (EhCache 3, Hazelcast, Infinispan, etc)
* [EhCache 2.x](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-ehcache2)
* [Hazelcast](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-hazelcast)
* [Infinispan](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-infinispan)
* [Couchbase](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-couchbase)
* [Redis](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-redis)
* [Caffeine](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-caffeine)
* [Guava](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-guava) (deprecated)
* [Simple](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-simple)

|  |
| --- |
| [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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-none) 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 theCacheManagerCustomizer 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. |

**31.1.1 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.

**31.1.2 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-hazelcast). 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.CacheManagerimplementation that the abstraction expects. No further customization is applied on it. |

**31.1.3 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 thespring-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

**31.1.4 Hazelcast**

Spring Boot has a [general support for Hazelcast](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-hazelcast). If a HazelcastInstance has been auto-configured, it is automatically wrapped in a CacheManager.

**31.1.5 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. |

**31.1.6 Couchbase**

If the Couchbase java client and the couchbase-spring-cache implementation are available and Couchbase is [configured](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-couchbase), 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.

**31.1.7 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. |

**31.1.8 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.

**31.1.9 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.

**31.1.10 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.

**31.1.11 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

**32. 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.

**32.1 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/4.3.11.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.

**32.1.1 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 inapplication.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/v1.5.7.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.

**32.1.2 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 inapplication.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 typeorg.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/v1.5.7.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.

**32.1.3 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

**32.1.4 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/4.3.11.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. |

**32.1.5 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/4.3.11.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) {

*// ...*

}

}

**32.2 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 thespring-boot-starter-amqp ‘Starter’.

**32.2.1 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 inapplication.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/v1.5.7.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](https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/) for more details. |

**32.2.2 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.

**32.2.3 Receiving a message**

When the Rabbit infrastructure is present, any bean can be annotated with @RabbitListener to create a listener endpoint. If no RabbitListenerContainerFactoryhas 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 aSimpleRabbitListenerContainerFactoryConfigurer 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] | **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. |

**32.3 Apache Kafka Support**

[Apache Kafka](https://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 inapplication.properties:

spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.consumer.group-id=myGroup

See [KafkaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/kafka/KafkaProperties.java) for more of the supported options.

**32.3.1 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;

}

*// ...*

}

**32.3.2 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) {

*// ...*

}

}

**32.3.3 Additional Kafka Properties**

The properties supported by auto configuration are shown in [Appendix A, *Common application properties*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties). 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);

}

}

**33. 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(). |

**33.1 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 aRestTemplateBuilder and will prevent any RestTemplateCustomizer beans from being used.

**34. 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) {

...

}

}

**35. 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/4.3.11.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/v1.5.7.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

**36. 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.

**36.1 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/1.5.7.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. |

**36.2 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. |

**36.3 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/1.5.7.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. |

**36.4 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-connecting-to-a-jndi-datasource) to configure your DataSource.

**36.5 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;

**36.6 Supporting an alternative embedded transaction manager**

The [XAConnectionFactoryWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/XAConnectionFactoryWrapper.java) and [XADataSourceWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/bitronix/BitronixXAConnectionFactoryWrapper.java) and [BitronixXADataSourceWrapper](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/bitronix/BitronixXADataSourceWrapper.java) provide good examples of how to write XA wrappers.

**37. 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-caching-provider-hazelcast). The HazelcastInstance is automatically wrapped in a CacheManager implementation if caching is enabled. |

**38. 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/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/integration/IntegrationAutoConfiguration.java) class for more details.

**39. 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/v1.5.7.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

**40. 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/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jmx/JmxAutoConfiguration.java) class for more details.

**41. 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.

**41.1 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#integration-testing) & Spring Boot Test — Utilities and integration test support for Spring Boot applications.
* [AssertJ](https://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.

**41.2 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#testing) of the Spring Framework reference documentation.

**41.3 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. |

**41.3.1 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 @Configurationto 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-structuring-your-code) 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. |

**41.3.2 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-detecting-config), @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/1.5.7.RELEASE/api/org/springframework/boot/context/TypeExcludeFilter.html)for details. |

**41.3.3 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-discover-the-http-port-at-runtime) 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");

}

}

**41.3.4 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 @Configurationclasses 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.

### 41.3.5 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. |

### 41.3.6 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration).

### 41.3.7 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration).

### 41.3.8 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/4.3.11.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/v1.5.7.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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration).

### 41.3.9 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/4.3.11.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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration).

### 41.3.10 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration).

### 41.3.11 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 componentsattribute 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#test-auto-configuration).

### 41.3.12 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}");

}

}

### 41.3.13 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-springintegrates 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-dependency-versions) in your build.gradle or pom.xml file. |

When using Spock 1.1, the annotations [described above](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications) 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;

}

}

## 41.4 Test utilities

A few test utility classes are packaged as part of spring-boot that are generally useful when testing your application.

### 41.4.1 ConfigFileApplicationContextInitializer

ConfigFileApplicationContextInitializer is an ApplicationContextInitializer that can apply to your tests to load Spring Boot application.propertiesfiles. 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. |

### 41.4.2 EnvironmentTestUtils

EnvironmentTestUtils allows you to quickly add properties to a ConfigurableEnvironment or ConfigurableApplicationContext. Simply call it withkey=value strings:

EnvironmentTestUtils.addEnvironment(env, "org=Spring", "name=Boot");

### 41.4.3 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"));

}

}

### 41.4.4 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", 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(...);

}

}

}

## 42. 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#websocket) that can be easily accessed via the spring-boot-starter-websocket module.

## 43. 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.

## 44. 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter).

|  |
| --- |
| [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. |

## 44.1 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/v1.5.7.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/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/resources/META-INF/spring.factories) file).

## 44.2 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 theEnableAutoConfiguration 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/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/AutoConfigureAfter.java) or [@AutoConfigureBefore](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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. |

## 44.3 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 @Beanmethods.

### 44.3.1 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. |

### 44.3.2 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. |

### 44.3.3 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.

### 44.3.4 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.

### 44.3.5 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.

### 44.3.6 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/4.3.11.RELEASE/spring-framework-reference/htmlsingle/#expressions).

## 44.4 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. |

### 44.4.1 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 starteracme-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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-annotation-processor) 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.

### 44.4.2 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. |

### 44.4.3 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.

## 45. 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 Boot API documentation](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api) or you can browse the [source code directly](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE). If you have specific questions, take a look at the [how-to](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto) section.

If you are comfortable with Spring Boot’s core features, you can carry on and read about [production-ready features](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready).

# 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.](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-actuator-with-jersey)

## 46. Enabling production-ready features

The [spring-boot-actuator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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")

}

## 47. 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 | Provides a hypermedia-based “discovery page” for the other endpoints. Requires Spring HATEOAS to be on the classpath. | true |
| auditevents | Exposes audit events information for the current application. | 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 | Performs a thread dump. | true |
| env | Exposes properties from Spring’s ConfigurableEnvironment. | 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 | Displays arbitrary application info. | false |
| loggers | Shows and modifies the configuration of loggers in the application. | 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 |
| shutdown | Allows the application to be gracefully shutdown (not enabled by default). | true |
| trace | Displays trace information (by default the last 100 HTTP requests). | true |

If you are using Spring MVC, the following additional endpoints can also be used:

| **ID** | **Description** | **Sensitive Default** |
| --- | --- | --- |
| docs | Displays documentation, including example requests and responses, for the Actuator’s endpoints. Requires spring-boot-actuator-docs to be on the classpath. | 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). |

## 47.1 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

## 47.2 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.

## 47.3 CORS support

[Cross-origin resource sharing](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing) (CORS) is a [W3C specification](https://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 GETand POST calls from the example.com domain:

endpoints.cors.allowed-origins=http://example.com

endpoints.cors.allowed-methods=GET,POST

|  |
| --- |
| [Tip] |
| Check [EndpointCorsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointCorsProperties.java) for a complete list of options. |

## 47.4 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. |

## 47.5 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”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-health-access-restrictions)for HTTP details).

Health information is collected from all [HealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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-configuredHealthIndicators 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.

## 47.6 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.

### 47.6.1 Auto-configured HealthIndicators

The following HealthIndicators are auto-configured by Spring Boot when appropriate:

| **Name** | **Description** |
| --- | --- |
| [CassandraHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/CassandraHealthIndicator.java) | Checks that a Cassandra database is up. |
| [DiskSpaceHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/DiskSpaceHealthIndicator.java) | Checks for low disk space. |
| [DataSourceHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/DataSourceHealthIndicator.java) | Checks that a connection to DataSource can be obtained. |
| [ElasticsearchHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/ElasticsearchHealthIndicator.java) | Checks that an Elasticsearch cluster is up. |
| [JmsHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/JmsHealthIndicator.java) | Checks that a JMS broker is up. |
| [MailHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/MailHealthIndicator.java) | Checks that a mail server is up. |
| [MongoHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/MongoHealthIndicator.java) | Checks that a Mongo database is up. |
| [RabbitHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/RabbitHealthIndicator.java) | Checks that a Rabbit server is up. |
| [RedisHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/RedisHealthIndicator.java) | Checks that a Redis server is up. |
| [SolrHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/SolrHealthIndicator.java) | Checks that a Solr server is up. |
| [Tip] |
| It is possible to disable them all using the management.health.defaults.enabled property. |

### 47.6.2 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/v1.5.7.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/v1.5.7.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/v1.5.7.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 HttpStatus.SERVICE\_UNAVAILABLE:

endpoints.health.mapping.FATAL=503

The default status mappings for the built-in statuses are:

| **Status** | **Mapping** |
| --- | --- |
| DOWN | SERVICE\_UNAVAILABLE (503) |
| OUT\_OF\_SERVICE | SERVICE\_UNAVAILABLE (503) |
| UP | No mapping by default, so http status is 200 |
| UNKNOWN | No mapping by default, so http status is 200 |

## 47.7 Application information

Application information exposes various information collected from all [InfoContributor](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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.

### 47.7.1 Auto-configured InfoContributors

The following InfoContributors are auto-configured by Spring Boot when appropriate:

| **Name** | **Description** |
| --- | --- |
| [EnvironmentInfoContributor](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/EnvironmentInfoContributor.java) | Expose any key from the Environment under the info key. |
| [GitInfoContributor](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/GitInfoContributor.java) | Expose git information if a git.properties file is available. |
| [BuildInfoContributor](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/BuildInfoContributor.java) | Expose 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. |

### 47.7.2 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-automatic-expansion).  Assuming you are using Maven, you could rewrite the example above as follows:  info.app.encoding=@project.build.sourceEncoding@  info.app.java.source=@java.version@  info.app.java.target=@java.version@ |

### 47.7.3 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 aGitProperties 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-git-info) 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

### 47.7.4 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-build-info) for more details. |

### 47.7.5 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/v1.5.7.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"

**}**

**}**

## 48. 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.

## 48.1 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 standardHttpServletRequest.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. |

## 48.2 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 port](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-management-server-port), 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. |

## 48.3 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.)

## 48.4 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

## 48.5 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

## 48.6 Disabling HTTP endpoints

If you don’t want to expose endpoints over HTTP you can set the management port to -1:

management.port=-1

## 48.7 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, setendpoints.health.sensitive to true. Here’s a summary of behavior (with default sensitive flag value “false” indicated in bold):

| **management.security.enabled** | **endpoints.health.sensitive** | **Unauthenticated** | **Authenticated (with right role)** |
| --- | --- | --- | --- |
| false | \* | Full content | Full content |
| true | **false** | Status only | Full content |
| true | true | No content | Full content |

## 49. 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.

## 49.1 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

## 49.2 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

## 49.3 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.

### 49.3.1 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

### 49.3.2 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

## 50. 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. |

## 50.1 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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security) 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/).

$ ssh -p 2000 user@localhost

user@localhost's password:

. \_\_\_\_ \_ \_\_ \_ \_

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

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

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

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

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

:: Spring Boot :: (v1.5.7.RELEASE) on myhost

Type help for a list of commands. Spring Boot provides metrics, beans, autoconfig and endpoint commands.

### 50.1.1 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/1.5.7.RELEASE/api/org/springframework/boot/actuate/autoconfigure/CrshAutoConfiguration.html) and [ShellProperties](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/actuate/autoconfigure/ShellProperties.html)Javadoc for full details.

## 50.2 Extending the remote shell

The remote shell can be extended in a number of interesting ways.

### 50.2.1 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 |
| spring.beanfactory | Access to the Spring BeanFactory |
| spring.environment | Access to the Spring Environment |

### 50.2.2 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/).

## 51. 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.

## 51.1 Configure a Logger

In order to configure a given logger, you POST a partial entity to the resource’s URI:

**{**

"configuredLevel": "DEBUG"

**}**

## 52. 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/v1.5.7.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/v1.5.7.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) 4times. 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. |

## 52.1 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)

## 52.2 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 isdatasource.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.

## 52.3 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.

## 52.4 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.

## 52.5 Recording your own metrics

To record your own metrics inject a [CounterService](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/GaugeService.java) into your bean. The CounterService exposes increment, decrement and resetmethods; 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/). |

## 52.6 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.

## 52.7 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. |

## 52.8 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. |

### 52.8.1 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. |

### 52.8.2 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](http://localhost:4242/api/put)). It also has a namingStrategythat 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](http://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

}

}

]

### 52.8.3 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);

}

### 52.8.4 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).

## 52.9 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. |

## 52.10 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).

## 52.11 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/1.5.7.RELEASE/api/org/springframework/boot/actuate/metrics/writer/Delta.html) or [Metric](http://docs.spring.io/spring-boot/docs/1.5.7.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).

## 53. 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).

## 54. 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** | **Description** |
| --- | --- |
| Request Headers | 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. |

## 54.1 Custom tracing

If you need to trace additional events you can inject a [TraceRepository](https://github.com/spring-projects/spring-boot/tree/v1.5.7.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.

## 55. 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 nameapplication.port).

These writers are not activated by default, but you can enable them in one of the ways described below.

## 55.1 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

## 55.2 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.

## 56. Cloud Foundry support

Spring Boot’s actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The /cloudfoundryapplicationpath 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. |

## 56.1 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

## 56.2 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

## 56.3 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();

}

## 57. 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/v1.5.7.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’](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment) or jump ahead for some in-depth information about Spring Boot’s [*build tool plugins*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins).