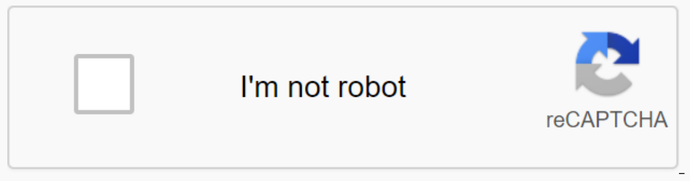

Continue

Spring boot documentation pdf download

Spring Boot helps you build independent production grade spring-based applications that you can run. We have an opinion of the Spring platform and third-party libraries, so you can start with the slightest fuss. Most Spring Boot applications require very little Spring configuration. You can use Spring Boot to create Java applications that can be started using java -jar or more traditional war implementations. We also provide a command-line tool that runs spring scripts. Provide a radically faster and more widely accessible introductory experience for all spring development. Be reviewed out of the box, but get out of the way quickly as the requirements begin to diverge from the default values. Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and outsourced configuration). Absolutely no code generation and no requirement for XML configuration. The reference documentation consists of the following sections: Legal information. Overview of documentation on documentation, getting help, getting started, and more. Introduction to Spring Boot, system requirements, Servlet containers, Spring Boot installation, development of your first spring boot application with spring Boot Build Systems, code structuring, configuration, Spring Beans and dependency injection, DevTools and more. Spring Boot features profiles, logging, security, caching, spring integration, testing, and more. Spring start actuator monitoring, metrics, auditing and more. Deploying Spring Resort Boot Applications Cloud Deployment, Installing as a Unix Application. Spring Boot CLI Installing the CLI, using the CLI, configuring the CLI, and more. Build tools plugins Maven Plugin, Gradle Plugin, Antlib, and more. How to do this Application Development Guides, configuration, built-in servers, data access, and more. The reference documentation has the following appendices: This section goes into the details of Spring Boot. Here you can learn about the key features you can use and customize. If you haven't already, you may want to read the Getting Started and Using Spring Boot sections, so you have a good grounding of the basics. The SpringApplication class provides a convenient way to boot a Spring application that starts from a main() method. In many situations, you can delegate to the static SpringApplication.run method, as shown in the following example: public static void main(String[] args) ? SpringApplication.run(MySpringConfiguration.class, args); When the you should see something similar to the following output: . ____ _ __ _ _ /\\ / ___'_ __ _ _(_)_ __ __ _ \ \ \ \ (( )___ | '_ | '_| | '_ \/ _` | \ \ \ \ \\/ ___)| |_)| | | | | || (_| |)))) ' |____| .__|_| |_|_| |___, | / / / / á :: Boot Boot :: v2.3.4.RELEASE 2019-04-31 13:09:54.117 INFO 56603 --- [[o.s.b.s.app.SampleApplication : Starting SampleApplication v0.1.0 on my computer with PID 56603 (/apps/myapp.jar started by pwebb) 2019-04-31 13:09:54.166 INFO 56603 --- [main] ationConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.ser6e5a8246: start date [Wed Jul 31 00:08:16 PDT 2013]; context hierarchy root 2019-04-01 13:09:56.912 INFO 41370 --- [main] .t.TomcatServletWebServerFactory : Server initialized with port: 8080 2019-04- 13:09:57.501 INFO 41370 --- [main] o.s.app.SampleApplication : SampleApplication started in 2,992 seconds (JVM running for 3,658) By default, INFO log messages are displayed, including some relevant start details, as the user who started the application. If you need a logging level other than INFO, you can set it, as described in Posting levels. The application version is determined by the package deployment version of the main application class. The log of startup information can be disabled by setting spring.main.log-startup-info to false. This will also disable registration of active profiles in the application. To add an additional record during startup, you can override logStartupInfo(boolean) in a SpringApplication subclass. If the application does not start, registered FailureAnalyzers have the opportunity to provide a dedicated error message and a specific action to fix the problem. For example, if you start a web application on port 8080 and that port is already in use, you should see something similar to the following message: ********** Port 8080 was already in use. Action: Identify and stop the process you are listening for on port 8080 or configure this application to listen on another port. Spring Boot provides numerous implementations of FailureAnalyzer and you can add your own. If no error parser can handle the exception, you can still display the full condition report to better understand what went wrong. To do this, you must enable the debug property or enable DEBUG logging for org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener. For example, 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 SpringApplication allows you to initialize an application lazily. When lazy initialization is enabled, beans are created as needed instead of during application startup. As a result, enabling initialization you can reduce the time it takes for your application to start. In a web application, when you enable lazy initialization, many web-related beans will not be initialized until an HTTP request is received. One disadvantage of lazy initialization is that it can delay detecting a problem with your application. If a misconfigured bean is lazily initialized, a it will no longer occur during startup and the problem will only become apparent when the bean is initialized. Care should also be taken to ensure that the JVM has enough memory to accommodate all the beans in the application and not just those that are initialized during startup. For these reasons, lazy initialization is not enabled by default, and it is recommended that fine-tuning the JVM heap size be done before you enable lazy initialization. Lazy initialization can be enabled programmatically by using the lazyInitialization method in SpringApplicationBuilder or the setLazyInitialization method in SpringApplication. Alternatively, it can be enabled by using the spring.main.lazy-initialization property as shown in the following example: spring.main.lazy-initialization-true If you want to disable lazy initialization for certain beans while using lazy initialization for the rest of your application, you can explicitly set its deferred attribute to false by using the @Lazy(false) annotation. The banner that is printed at startup can be changed by adding a banner.txt file to its class path or by setting the spring.banner.location property to the location of that file. If the file is encoded other than UTF-8, you can set spring.banner.charset. In addition to a text file, you can also add a banner.gif, banner.jpg, or banner.png image file to the class path or set the spring.banner.image.location property. Images are converted to an ASCII art representation and printed on top of any text banner. Within your banner.txt file, you can use any of the following placeholders: Table 4. Banner variables Description of the variable $-application.version- The version number of the application, as declared in MANIFEST. Mf. For example, Deployment-Version: 1.0 is printed as 1.0. $-application.formatted-version – The application version number, as declared in MANIFEST. MF and formatted for display (surrounded by square brackets and preset with v). For example (v1.0). $-spring-boot.version- The version of Spring Boot you are using. For example 2.3.4.RELEASE. $-spring-boot.formatted-version- The version of Spring Boot you are using, formatted for display (surrounded by square brackets and prefixed with v). For example (v2.3.4.RELEASE). $'Ansi.NAME' (or $'AnsiColor.NAME', '$'AnsiBackground.NAME', '$'AnsiStyle.NAME') Where NAME is the name of an ANSI escape code. See AnsiPropertySource for more information. The title of the application, as declared in MANIFEST. Mf. For example, Implementation-Title: MyApp is printed as MyApp. 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 printBanner() method. You can also use the spring.main.banner-mode property to determine whether the banner should be printed in System.out (console), sent to the configured logger (log), or not occur at all (off). The printed banner is registered as a singleton singleton bean the following name: springBootBanner. If springApplication defaults are not to your liking, you can create a local instance and customize it. For example, to disable the banner, you can type: public static void main(String[] args) ? SpringApplication app ? new SpringApplication(MySpringConfiguration.class); app.setBannerMode(Banner.Mode.OFF); app.run(args); Constructor arguments passed to SpringApplication are configuration sources for spring beans. In most cases, these are references to classes @Configuration, but they could also be references to the XML configuration or packages to examine. You can also configure SpringApplication by using an application.properties file. See Outsourced Configuration for more information. If you need to create an ApplicationContext hierarchy (multiple contexts with a parent/child relationship) or if you prefer to use a fluid factory API, you can use SpringApplicationBuilder. SpringApplicationBuilder allows you to chain multiple method calls and includes parent and child methods that allow you to create a hierarchy, as shown in the following example: new SpringApplicationBuilder() .sources(Parent.class) .child(Application.class) .bannerMode(Banner.Mode.OFF) .run(args); There are some restrictions when creating an ApplicationContext hierarchy. For example, Web components must be contained in the child context and the same environment is used for the parent and child contexts. See SpringApplicationBuilder Javadoc for full details. When deployed on platforms, applications can provide information about their availability to the platform using infrastructure such as Kubernetes Probes. Spring Boot includes out-of-the-box support for lively availability and commonly used readiness states. If you are using Spring Boot actuator support, these states are exposed as maintenance endpoint groups. In addition, you can also get availability states by injecting the ApplicationAvailability interface into your own beans. The Liveness state of an application indicates whether its internal state allows it to function properly or recover by itself if it is currently failing. A broken Liveness state means that the application is in a state from which it cannot be recovered and the infrastructure must restart the application. In general, the Vivaz state should not be based on external checks, such as health checks. Doing so would trigger a failed external system (a database, a web API, an external cache) would trigger massive restarts and cascading errors across the platform. The internal state of Spring Boot applications is primarily represented by Spring ApplicationContext. If the application context has started successfully, Spring Boot assumes that the application is in a valid state. application is considered active as soon as the context has been updated, see Spring Boot Application Lifecycle and Related Application Events. The status of preparation of an application if the app is ready to handle traffic. A Failed Preparation status tells the platform not to route traffic to the application for now. This usually occurs during startup, while the CommandLineRunner and ApplicationRunner components are processed, or at any time if the application decides it is too busy for additional traffic. The CommandLineRunner and ApplicationRunner components must run tasks that are expected to run during startup instead of

using Spring component lifecycle callbacks, such as @PostConstruct. Application components can retrieve the current availability state at any time by injecting the ApplicationAvailability interface and calling methods. More often, apps will want to listen to status updates or update app status. For example, we can export the Readiness state of the application to a file so that a Kubernetes exec Probe can look at this file: @Component public class ReadinessStateExporter - @EventListener public void onStateChange(AvailabilityChangeEvent<ReadinessState> event) - switch (event.getState()) - case ACCEPTING_TRAFFIC: // create /tmp/healthy break file; case REFUSING_TRAFFIC: // delete /tmp/healthy break file; We can also update the state of the application, when the application is interrupted and cannot be retrieved: @Component public class LocalCacheVerifier - private final ApplicationEventPublisher eventPublisher; Public LocalCacheVerifier(ApplicationEventPublisher eventPublisher) to this.eventPublisher á eventPublisher; ? public void checkLocalCache() to try ? //... ? catch (CacheCompletelyBrokenException ex) ? AvailabilityChangeEvent.publish(this.eventPublisher, ex, LivenessState.BROKEN); ? In addition to common Spring Framework events, such as ContextRefreshedEvent, a SpringApplication sends some additional application events. Some events are actually raised before you create ApplicationContext, so you cannot register a listener with them as a @Bean. You can register them with the SpringApplication.addListeners(...) method or the SpringApplicationBuilder.listeners(... method). If you want those listeners to register automatically, regardless of how the application is created, you can add a META-INF/spring.factories file to your project and reference the listeners using the org.springframework.context.ApplicationListener key, As shown in the following example: org.springframework.context.ApplicationListener-com.example.project.MyListener Application events are sent in the following order, as the application runs: an ApplicationStartingEvent is sent at the start of an execution, but before any processing, except for logging listeners and initializers. A sent when the environment to be used in the context is known, but before the context is created. An ApplicationContextInitializedEvent is sent when the ApplicationContext is ready and ApplicationContextInitializers have been </ReadinessState> </ReadinessState> but before the bean definitions are loaded. An ApplicationPreparedEvent is sent just before the update starts, but after the bean definitions have been loaded. An ApplicationStartedEvent is sent after the context has been updated, but before any application and command-line corridors have been called. An AvailabilityChangeEvent is sent immediately afterwards with LivenessState.CORRECT to indicate that the application is considered live. An ApplicationReadyEvent is sent after any application and command-line brokers have been called. An AvailabilityChangeEvent is sent immediately after ReadinessState.ACCEPTING_TRAFFIC to indicate that the application is ready to handle requests. An ApplicationFailedEvent is sent if there is an exception at startup. The above list only includes SpringApplicationEvents that are linked to SpringApplication. In addition to these, the following events are also published after ApplicationPreparedEvent and before ApplicationStartedEvent: A WebServerInitializedEvent is sent after the WebServer is ready. ServletWebServerInitializedEvent and ReactiveWebServerInitializedEvent are the servlet and reactivate variants respectively. A ContextRefreshedEvent is sent when an ApplicationContext is updated. You often don't need to use application events, but it can be helpful to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks. Application events are sent using the Spring Framework event publishing mechanism. Part of this mechanism ensures that an event published to listeners in a secondary context is also published to listeners in any predecessor context. As a result, if your application uses a SpringApplication instance hierarchy, a listener can receive multiple instances of the same application event type. To allow the listener to distinguish between an event for its context and an event for a descendant context, you must request that its application context be injected and then compare the inline context with the event context. The context can be inserted by implementing ApplicationContextAware or, if the listener is a bean, by using @Autowired. A SpringApplication attempts to create the correct Type of ApplicationContext on your behalf. The algorithm used to determine a WebApplicationType is as follows: If Spring MVC is present, an AnnotationConfigServletWebServerApplicationContext is used if Spring MVC is not present and Spring WebFlux is present, an AnnotationConfigReactiveWebServerApplicationContext is used otherwise, AnnotationConfigApplicationContext uses this means that if you use Spring MVC and the new Spring WebFlux WebClient in the same application, Spring MVC will be used otherwise You can easily override it by calling setWebApplicationType(WebApplicationType). It is also possible to take full control of the ApplicationContext type that is used by calling setApplicationContextClass(...). It is often desirable to call SpringApplication is used within a JUnit test. If you need to access the application arguments that were passed to SpringApplication.run(...)), you can insert an org.springframework.boot.ApplicationArguments bean. The ApplicationArguments interface provides access to both raw String[] arguments and option and non-optional arguments, as shown in the following example: import org.springframework.boot.*; import org.springframework.beans.factory.annotation.*; import org.springframework.stereotype.*; @Component public class MyBean a @Autowired MyBean(ApplicationArguments args) to boolean debug a args.containsOption(debug); <String> List Files: args.getNonOptionArgs(); If running with --debug logfile.txt debug-true, files-[logfile.txt] - Spring Boot also registers a CommandLinePropertySource with spring Environment. This also allows you to insert single-application arguments using @Value. If you need to run specific code after SpringApplication has started, you can implement the ApplicationRunner or CommandLineRunner interfaces. Both interfaces work in the same way and offer a single execution method, which is called just before SpringApplication.run(...) completes. This contract is appropriate for tasks that must run after application startup, but before it starts accepting traffic. CommandLineRunner interfaces provide access to application arguments as an array of strings, while ApplicationRunner uses the ApplicationArguments interface described above. The following example shows a CommandLineRunner with a run method: import org.springframework.boot.*; import org.springframework.stereotype.*; @Component public class MyBean implements CommandLineRunner - public void run(String... args) to // Do something... If you define multiple CommandLineRunner or ApplicationRunner beans to call in a specific order, you can also implement the org.springframework.core.Ordered interface or use the org.springframework.core.annotation.Order annotation. Each SpringApplication registers a shutdown hook with the JVM to ensure that ApplicationContext closes successfully on exit. All standard Spring lifecycle callbacks (such as the DisposableBean interface or annotation) can @PreDestroy). In addition, beans can implement the org.springframework.boot.ExitCodeGenerator interface if they want to return a specific exit code when SpringApplication.exit() is called. This exit code can be passed to System.exit() to return it as a status code, as shown in the following example: @SpringBootApplication ExitCodeApplication public class? @Bean exitCodeGenerator exitCodeGenerator() to return() -> 42; ? public static void main(String[] args) ? args)); In addition, the ExitCodeGenerator interface can be implemented by exceptions. When such an exception is encountered, Spring Boot returns the output </String> </String> provided by the implemented getExitCode() method. You can enable administrator-related features for your application by specifying the spring.application.admin.enabled property. This exposes springApplicationAdminMXBean on the MBeanServer platform. You can use this feature to manage the Spring Boot application remotely. This feature might also be useful for any service container deployment. If you want to know which HTTP port your application is running on, get the property with a key from local.server.port. Spring Boot allows you to outsource settings so you can work with the same application code in different environments. You can use property files, YAML files, environment variables, and command-line arguments to outsource settings. Property values can be inserted directly into beans using @Value annotation, which is accessed through Spring environment abstraction or can be bound to structured objects through @ConfigurationProperties. Spring Boot uses a very particular PropertySource order that is designed to allow sensible override of values. The properties are considered in the following order: Devtools global configuration properties in the $HOME/.config/spring-boot directory when devtools is active. @TestPropertySource annotations in the tests. properties in testing. Available in @SpringBootTest and test annotations to test a particular slice of your application. Command-line arguments. System properties SPRING_APPLICATION_JSON (inline JSON embedded in an environment variable or system property). Init ServletConfig parameters. ServletContext init parameters. JNDI from java:comp/env. Java System Properties (System.getProperties()). Operating system environment variables. A RandomValuePropertySource that has properties only in random.*. Profile-specific application properties outside the packaged jar (application-profile-.properties and YAML variants). Profile-specific application properties packaged within your jar (application-profile-.properties and YAML variants). Application properties outside the packaged jar (application.properties and YAML variants). Application properties packaged within the jar (application.properties and YAML variants). @PropertySource annotations in the @Configuration. Note that these property sources are not added to the environment until the application context is updated. This is too late to configure certain properties such as logging.* and spring.main.* that are read before the upgrade begins. Default properties (specified by setting SpringApplication.setDefaultProperties). To provide a concrete example, let's say you develop @Component that a name property, as shown in the following example: import org.springframework.stereotype.*; import org.springframework.beans.factory.annotation.*; @Component public class MyBean a @Value($-name-) private string name; // ... In the application class path (for example, inside your jar) jar) you can have an application.properties file that provides a reasonable default property value for name. When running in a new environment, an application.properties file can be provided outside the jar file that replaces the name. For one-time testing, you can start with a specific command-line switch (for example, java -jar app.jar --name-Spring). Spring Boot also supports wildcard locations when uploading configuration files. By default, a wildcard location of config/*/ outside the jar is supported. Wildcard locations are also supported when specifying spring.config.additional-location and spring.config.location. Wildcard locations are especially useful in an environment such as Kubernetes when there are multiple configuration property sources. For example, if you have any Redis settings and any MySQL settings, you might want to keep those two configuration items separate, while requiring both to be present in an application.properties to which the application can bind. This could result in two separate application.properties files mounted in different locations, such as /config/redis/application.properties and /config/mysql/application.properties. In such a case, having a wildcard location of config/*/will result in both files being processed. A wildcard location must contain only one and end with / for search locations that are directories or */<filename> for search locations that are files. Wildcard locations are sorted alphabetically based on the absolute path of the file names. The properties SPRING_APPLICATION_JSON can be provided on the command line with an environment variable. For example, you could use the following line in an UN*X shell: $SPRING_APPLICATION_JSON"acme:'name:test" java -jar myapp.jar In the example above, it ends with acme.name-test in the spring environment. You can also provide JSON such as spring.application.json in a System property, as shown in the following example: $ java -Dspring.application.json-'"name:test" -jar myapp.jar You can also provide the JSON using a command-line argument, as shown in the following example: $ java -jar myapp.jar --spring.application.json'"name:test" You can also provide the JSON as a JNDI variable, as follows: java:comp/env/spring.application. Although null values in the JSON will be added to the source of the resulting property, PropertySourcesPropertyResolver treats null properties as missing values. This means that JSON cannot override properties of lower-order property sources with a null value. RandomValuePropertySource is useful for inserting random values (for example, in secrets or test cases). It can produce integers, longs, uuids, or as shown in the following example: my.secret-$-$-$ random.value, my.number, $, random.int, my.bignumber, $, my.uuid, $, my.u.number, my.number.less.than.ten, $, etc., my.number.n., in.range, $, etc., int[1024,65536], the syntax random.int* is the value OPEN (, max) CLOSE </filename> </filename gt; OPEN,CLOSE are any character and value, max are integers. If max is provided, the value is the minimum value and max is the maximum (exclusive) value. By default, SpringApplication converts any command-line option arguments (that is, arguments that begin with --, such as --server.port-9000) to a property and adds them to the spring environment. As mentioned earlier, command-line properties always take precedence over other property sources. If you do not want command-line properties to be added to your environment, you can disable them by using SpringApplication.setAddCommandLineProperties(false). SpringApplication loads application.properties file properties into the following locations and adds them to the spring environment: A /config subdirectory of the current directory The classpath /config Class Path Package The class path root The list is sorted by precedence (properties defined at higher locations in the list override those defined in lower locations). You can also use YAML files ('.yml') as an alternative to '.properties'. If you do not like application.properties as the configuration file name, you can change it to another file name by specifying a spring.config.name. You can also reference an explicit location by using the spring.config.location environment property (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name: $ java -jar myproject.jar --spring.config.name-myproject The following example shows how to specify two locations: $java -jar myproject.jar --spring.config.location-classpath:/default.properties,classpath:/override.properties spring.config.name and spring.config.location are used very soon to determine which files should be loaded. They must be defined as an environment property (typically an operating system environment variable, a system property, or a command-line argument). If spring.config.location contains directories (unlike files), they must end in / (and, at run time, append with names generated from spring.config.name before loading, including profile-specific file names). The files specified in spring.config.location are used as is, without support for profile-specific variants, and are replaced by profile-specific properties. Whether you specify it directly or are in a directory, configuration files must include a file extension on your behalf. Typical extensions that are supported from the factory are .properties, .yaml, and .yml. Configuration locations are searched in reverse order. By default, configured locations are The resulting search order is as follows: file:./config/ file:./config/*/ file:./ classpath:/config/ classpath:/ When you configure custom configuration locations using spring.config.location, they override the default locations. For example, if spring.config.location is set to the value the search order becomes the following: file:./custom-config/ classpath:custom-config/ Alternatively, when custom configuration locations are configured using spring.config.additional-location, they are used in addition to the default locations. Additional locations are searched before the default locations. For example, if additional classpath locations are configured:/custom-config/,file:./custom-config/, the search order becomes the following: file:./custom-config/ classpath:custom-config/ file:./config/file:./config/*/ file:./ classpath:/config/classpath:/ This search order allows you to specify default values in one configuration file, and then selectively override those values in another. You can provide default values for your application in application.properties (or any other base name you choose with spring.config.name) in one of the default locations. These default values can be overridden at run time with a different file located in one of the custom locations. If you use environment variables instead of system properties, most operating systems do not allow period-separated key names, but you can use underscores instead (for example, SPRING_CONFIG_NAME instead of spring.config.name). See Binding from Environment Variables for more information. If your application is running in a container, you can use JNDI properties (in java:comp/env) or servlet context initialization parameters instead of environment variables or system properties, or. In addition to application.properties files, profile-specific properties can also be defined using the following naming convention: application--profile-.properties. The environment has a set of default profiles (by default, [default]) that are used if no active profile is set. In other words, if no profile is explicitly activated, the properties of application-default.properties are loaded. Profile-specific properties are loaded from the same locations as standard application.properties, with profile-specific files that always replace non-specific ones, regardless of whether the profile-specific files are inside or outside the packaged jar. If multiple profiles are specified, a last profit strategy is applied. For example, profiles specified by the spring.profiles.active property are added after those configured through the SpringApplication API and therefore take precedence. If you specified files in spring.config.location, the profile-specific variants of those files are not taken into account. Use directories in spring.config.location if you also want to use properties specific to the Application.properties values are filtered through the existing environment when used, so you can reference values defined above (for example, from system properties). app.name-MyApp app.description-$-app.name is a spring Boot Spring Boot application does not provide any built-in support for encrypting property values, however, it provides the endpoints needed to modify the values in the Spring Environment. The EnvironmentPostProcessor interface allows you to manipulate the environment before the application starts. See Customize your environment or ApplicationContext before it starts for more information. If you are looking for a secure way to store credentials and passwords, the Spring Cloud Vault project provides support for storing outsourced settings in HashiCorp Vault. YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data. The SpringApplication class automatically supports YAML as an alternative to properties as long as it has the SnakeYAML library in the class path. If you use Starters, SnakeYAML is automatically provided by spring-boot-starter. Spring Framework provides two convenient classes that can be used to load YAML documents. The YamlPropertiesFactoryBean loads YAML as Properties and yamlMapFactoryBean loads YAML as a map. For example, consider the following YAML document: environments: dev: url: name: Developer Setup prod: url: url: name: My Cool App The previous example would be transformed into the following properties: environments.dev.url.url.url. environments.dev.name-Developer Setup environments.prod.url- environments.prod.name-My Cool App YAML lists are represented as property keys with dereferencers [index]. For example, consider the following YAML: my: servers: - dev.example.com - another.example.com The previous example would be transformed into these properties: my.servers[0]-dev.example.com my.servers[1]-another.example.com To bind to properties such as that by using Spring Boot Binder utilities (which is what @ConfigurationProperties does), you must have a property in the target bean of type java.util.List (or Set) and you must provide a setter or initialize it with a mutable value. For example, the following example binds to the properties shown above: @ConfigurationProperties(prefix-my) public class Config - list servers<String> private - new ArrayList<String>(); public List<String> getServers() - return this.servers; ? The YamlPropertySourceLoader class can be used to expose YAML as a PropertySource in the spring environment. This allows you to @Value annotation with placeholder syntax to access YAML properties. You can specify multiple profile-specific YAML documents in a single file by using a spring.profiles key to indicate when the document is applied, as shown in the following example: server: address: 192.168.1.100 --- spring: profiles: development server: address: 127.0.0.1 --- spring: profiles: production & eu-central server: address: 192.168.1.120 In the example above, if the profile server.address property is 127.0.0.1. Similarly, if the production and eu-central profiles are active, the server.address property is 192.168.1.120. If development, production and eu-central</String> </String> </String> </String> are not enabled, so the property value is 192.168.1.100. Therefore, spring.profiles can contain a profile name (for example, production) or a profile expression. A profile expression allows you to express more complicated profile logic, for example, production &(eu-central - eu-west). Refer to the reference guide for more details. If none is explicitly active when the application context starts, the default profiles are activated. Therefore, in the following YAML, we set a value for spring.security.user.password that is only available in the default profile: server: port: 8000 --- spring: profiles: default security: user: password: weak Whereas, In the following example, the password is always set because it is not associated with any profile, and would have to be explicitly reset on all other profiles as needed: server: port: 8000 spring: security: user: password: weak Spring profiles designated by using the spring.profiles element can optionally be denied using the ! Character. If denied and non-negative profiles are specified for a single document, at least one unennied profile must match and no denied profile can match. YAML files cannot be loaded using @PropertySource. Therefore, in case you need to load values that way, you must use a properties file. Using multiple YAML document syntax in profile-specific YAML files can result in unexpected behavior. For example, consider the following settings in a file: application-dev.yml server: port: 8000 --- spring: profiles: profiles: !test security: user: secret If you run the application with the --spring.profiles.active-dev argument, security.user.password might be set to secret, but this is not the case. The nested document will be filtered because the main file is named application-dev.yml. It is already considered profile-specific and nested documents will be ignored. We recommend that you do not mix profile-specific YAML files and multiple YAML documents. Keep using only one of them. Using the @Value($-property-) annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or the data is hierarchical in nature. Spring Boot provides an alternative method for working with properties that allows strongly typed beans to govern and validate application settings. You can bind a bean by declaring standard JavaBean properties as shown in the following example: 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(acme) public class AcmeProperties - private habilitado; private InetAddress remoteAddress; private final Security security ? new Security(); public boolean isEnabled() ? ... á public void setEnabled(boolean enabled) . . { ... } public void setRemoteAddress(InetAddress remoteAddress) a ... - Public security getSecurity() - ... - public static class Security - private string user name; Private string password; Private List<String> roles ? new ArrayList <> (Collections.singleton(USER)); public String getUsername() ? ... a public void setUsername(String username) a ... a public String getPassword()<String> á<String> ... The previous POJO defines the following properties: acme.enabled, with a value of false by default. acme.remote-address, with a type that can be forced from String. acme.security.username, with a nested security object whose name is determined by the property name. In particular, the return type is not used at all there and could have been SecurityProperties. acme.security.password. acme.security.roles, with a String collection that has the default USER. Properties that map to @ConfigurationProperties classes available in Spring Boot, which are configured through property files, YAML files, environment variables, and so on, are public APIs, but the accessors (getters/setters) of the class itself are not designed to be used directly. This layout is based on a default empty constructor, and getters and setters are typically required because binding is through standard Java Beans property descriptors, as in Spring MVC. A setter can be omitted in the following cases: maps, as long as they are initialized, need a getter but not necessarily a setter because they can be mutated by the binder. Collections and arrays can be accessed through an index (typically with YAML) or through a single comma-separated value (properties). In the latter case, a setter is required. We always recommend adding a setter for these types. If you initialize a collection, make sure that it is not immutable (as in the previous example). If nested POJO properties (such as the Security field in the previous example) are initialized, a setter is not required. If you want the linker to create the instance on the fly using its default constructor, you need a setter. Some people use The Lombok Project to add getters and setters automatically. Ensure that Lombok does not generate any particular constructors for such a type, because the container automatically uses it to instantiate the object. Finally, only standard Java Bean properties are considered and binding is not supported in static properties. The example in the previous section can be rewritten immutably as shown in the following example: package com.example; import java.net.InetAddress; import java.util.List; import import org.springframework.boot.context.properties.ConstructorBinding; import org.springframework.boot.context.properties.bind.DefaultValue; @ConstructorBinding @ConfigurationProperties(acme)</String> </String> </String> </String> AcmeProperties class - private final boolean enabled; Private final InetAddress remoteAddress; Private final security security; AcmeProperties(Boolean enabled, InetAddress remoteAddress, Security) - this.enabled - enabled; this.remoteAddress - remoteAddress; this.security - security; a public boolean isEnabled() ? Private final string password; Final list roles<String> private; Public security (string user name, string password, @DefaultValue(USER) Roles<String> list) - this.username - username; this.password ; this.roles ? roles; a public String getUsername() á ... a public String<String> getPassword() . In this configuration, the @ConstructorBinding is used to indicate that constructor binding should be used. This means that the linker will expect to find a constructor with the parameters that you want to have bound. Nested members of a @ConstructorBinding class (such as Security in the previous example) will also be bound through its constructor. Default values can be specified by @DefaultValue and the same conversion service will be applied to force the String value to the target type of a missing property. By default, if there are no properties bound to Security, the AcmeProperties instance will contain a null value for security. If you want to return a non-null instance of Security even when there are no properties bound to it, you can use an empty @DefaultValue annotation to do so: package com.example; import java.net.InetAddress; import java.util.List; import org.springframework.boot.context.properties.ConfigurationProperties; import org.springframework.boot.context.properties.ConstructorBinding; import org.springframework.boot.context.properties.bind.DefaultValue; @ConstructorBinding @ConfigurationProperties(acme) public class AcmeProperties - private final boolean enabled; private final InetAddress remoteAddress; private final Security Security; public AcmeProperties(boolean enabled, InetAddress remoteAddress, @DefaultValue Security security) - this.enabled - enabled; this.remoteAddress - remoteAddress; this.security ? To use constructor binding, the class must be enabled by analyzing configuration properties or @EnableConfigurationProperties. You cannot use constructor binding with beans created by normal Spring mechanisms (for example, @Component beans, beans created through @Bean methods, or beans loaded with @Import) If you have more than one constructor for the class you can also use @ConstructorBinding directly in the constructor to bind. The use of java.util.Optional with @ConfigurationProperties is not recommended because it is primarily intended for use as a return type. As such, it is not suitable for injecting configuration properties. For consistency with properties of tipos, si lo hace</String> </String> </String> </String> an Optional property and has no value, null instead of an Optional void will be bound. Spring Boot provides infrastructure for binding @ConfigurationProperties register them as beans. You can enable class-by-class configuration properties or enable configuration property analysis that works similarly to component analysis. Sometimes classes annotated with @ConfigurationProperties may not be suitable for analysis, for example, if you are developing your own automatic configuration or if you want to conditionally enable them. In these cases, specify the list of types to process using the @EnableConfigurationProperties. This can be done in any @Configuration class, as shown in the following example: @Configuration(proxyBeanMethods - false) @EnableConfigurationProperties(AcmeProperties.class) public class MyConfiguration - - To use configuration property analysis, add the @ConfigurationPropertiesScan annotation to your application. Typically, it is added to the main application class that is annotated with @SpringBootApplication but can be added to any @Configuration class. By default, analysis will occur from the package of the class that declares the annotation. If you want to define specific packages to scan, you can do so as shown in the following example: @SpringBootApplication @ConfigurationPropertiesScan of the MyApplication public class (com.example.app, org.acme.another -) When the @ConfigurationProperties bean is registered by analyzing the configuration property or through @EnableConfigurationProperties, the bean has a conventional name: <prefix>- , where the environment key prefix specified in the <fqn> <prefix>annotation @ConfigurationProperties is the <fqn>full name of the bean. If the annotation does not provide a prefix, only the fully qualified bean name is used. The name of the bean in the previous example is acme-com.example.AcmeProperties. We recommend that @ConfigurationProperties only deal with the environment and, in particular, do not inject other grains from the context. For corner cases, you can use setter injection or any of the *Aware interfaces provided by the framework (such as EnvironmentAware if you need access to your environment). If you still want to inject other beans using the constructor, the configuration property bean must be annotated with @Component and use JavaBean-based property binding. This configuration style works particularly well with the springApplication external YAML configuration, as shown in the following example: a application.yml acme: remote-address: 192.168.1.1 security: username: administrator roles: - USER - ADMIN - additional configuration as needed To work with beans, You can insert them in the same way as any other bean, as shown in the following example: @Service public class MyService - private final acmeProperties properties; @Autowired public properties MyService(AcmeProperties) - this.properties -</fqn> </prefix> </fqn> </prefix> </prefix> • //... @PostConstruct public void openConnection() to Server Server - New Server(this.properties.getRemoteAddress()); // ... Using the @ConfigurationProperties also allows you to generate metadata files that IDEs can use to provide automatic completion of their own keys. See the appendix for more information. In addition to @ConfigurationProperties to annotate a class, you can also use it in @Bean methods. Doing so can be especially useful when you want to bind properties to third-party components that are outside the control. To configure a bean from the Environment properties, add @ConfigurationProperties to your bean record, as shown in the following example: @ConfigurationProperties(prefix ? another) @Bean AnotherComponent anotherComponent() . Any JavaBean property defined with the other prefix maps to that AnotherComponent bean similar to the AcmeProperties example above. Spring Boot uses some relaxed rules to bind environment properties to @ConfigurationProperties beans, so there does not need to be an exact match between the Environment property name and the bean property name. Common examples where this is useful include script-separated environment properties (for example, context-path binds to contextPath) and uppercase environment properties (for example, PORT is bound to port). As an example, consider the following @ConfigurationProperties: @ConfigurationProperties(prefix-acme.my-project.person) of the OwnerProperties - private String firstName public class; public String getFirstName() to return this.firstName; ? public void setFirstName(String firstName) - this.firstName ? With the above code, you can use all of the following property names: Table 5. Relaxed binding property note acme.my-project.person.first-name Kebab case, which is recommended for use in .properties and .yml files. acme.myProject.person.firstName Standard case syntax. acme.my_project.person.first_name Underscore, which is an alternative format for use in .properties and .yml files. ACME_MYPROJECT_PERSON_FIRSTNAME case format, which is recommended when using system environment variables. The prefix value for the annotation must be case-sensitive (lowercase and separated by -, such as acme.my-project.person). Table 6. Property Source Simple List Properties Files Camel case, kebab case, or notation subscore Standard list syntax using [ ] or comma-separated values YAML Files Camel case, kebab case, or underscore notation Standard YAML list syntax or comma-separated values Environment Variables Upper case format with underscore as the delimiter (see Binding from Environment Variables). Numeric values surrounded by underscores (see from environment variables)' Camel case system properties, kebab case, or notation subaja Standard list syntax using [ ] or comma-separated values We recommend that, where possible, properties be stored stored lowercase kebab format, such as my.property-name-acme. When binding to Map properties, if the key contains anything other than lowercase alphanumeric characters or -, you must use bracket notation to preserve the original value. If the key is not surrounded by [], non-alphanumeric or - characters are removed. For example, consider binding the following properties to a Map: acme: map: [/key1]: value1 [/key2]: value2 /key3:value3 The above properties will be bound to a Map with /key1, /key2, and key3 as keys in the map. For YAML files, square brackets must be surrounded by quotation marks for keys to be parsed correctly. Most operating systems impose strict rules around names that can be used for environment variables. For example, Linux shell variables can contain only letters (a to z or A to Z), numbers (0 to 9), or the underscore (_). By convention, Unix shell variables will also have their names in UPPERCASE. Spring Boot's relaxed binding rules are designed, as far as possible, to support these naming restrictions. To convert a property name canonically to an environment variable name, you can follow these rules: Replace points (.) with underscores (_). Remove the hyphens (-). Convert to uppercase. For example, the spring.main.log-startup-info configuration property would be an environment variable named SPRING_MAIN_LOGSTARTUPINFO. Underscores cannot be used to replace hyphens in property names. If you try SPRING_MAIN_LOG_STARTUP_INFO to the previous example, no value will be bound. Environment variables can also be used when binding to object lists. To bind to a list, the item number must be surrounded by underscores in the variable name. For example, the my.acme[0].other configuration property would use an environment variable named MY_ACME_0_OTHER. When lists are configured in more than one place, the override works by replacing the entire list. For example, suppose a MyPojo object with name and description attributes that are null by default. The following example exposes a list of AcmeProperties MyPojo objects: @ConfigurationProperties(acme) public class AcmeProperties - private final list<MyPojo> - new ArrayList <> (); public List<MyPojo> getList() - return this.list; ? Consider the following settings: acme: list: - name: my name description: my description --- spring: profiles: dev acme: list: - name: my another name If the dev profile is not active, AcmeProperties.list contains one MyPojo entry, as previousdefined. However, if the development profile is enabled, the list still includes only one entry (with a name of my other name and a description of This setting does not add a second instance of MyPojo to the list and does not merge the items. When you specify a list across multiple profiles, the one with the highest priority (and only that) is used. Consider the following example: acme: list: - name: my name description: my</MyPojo> </MyPojo> </MyPojo> - name: another name description: another description --- spring: profiles: dev acme: list: - name: my another name In the example above, if the dev profile is active, AcmeProperties.list contains a MyPojo entry (with a name of my other name and a description of null). For YAML, comma-separated lists and YAML lists can be used to completely replace the contents of the list. For Map properties, you can bind to property values extracted from multiple sources. However, for the same property across multiple sources, the one with the highest priority is used. The following example exposes a Map<String, mypojo-> from AcmeProperties: @ConfigurationProperties(acme) public class AcmeProperties - private final map<String, mypojo-> map - new HashMap <> (); public Map<String, mypojo-> getMap() - return this.map; ? Consider the following settings: acme: map: key1: name: my name 1 description: my description 1 --- spring: profiles: dev acme: map: key1: name: dev name 1 key2: name: dev name 2 description: dev description 2 If the dev profile is not active, AcmeProperties.map contains an entry with key1 (with a name of my name 1 and a description of my description 1). However, if the development profile is enabled, the map contains two entries with keys key1 (with a name dev name 1 and a description of my description 1) and key2 (with a name dev name 2 and a description of development description 2). The above merge rules apply to properties in all property sources and not just YAML files. Spring Boot attempts to force the properties of the external application to the correct type when bound to @ConfigurationProperties beans. If you need a custom type conversion, you can provide a ConversionService bean (with a bean named conversionService) or custom property editors (via a CustomEditorConfigurer bean) or custom converters (with bean definitions annotated as @ConfigurationPropertiesBinding). Because this bean is requested very soon during the application lifecycle, be sure to limit the dependencies that ConversionService uses. Typically, any dependencies you need may not be fully initialized at creation time. You may want to rename your custom ConversionService if it is not required for configuration key coertion and is only based on custom converters qualified with @ConfigurationPropertiesBinding. Spring Boot has dedicated support for expressing durations. If you expose a java.time.Duration property, the following formats are available in application properties: a normal long representation (using milliseconds as the default unit unless an @DurationUnit) The standard ISO-8601 format used java.time.Duration A more readable format where the value and drive are docked (for example. 10s means 10 seconds) Consider the following example: @ConfigurationProperties(app.system) public class AppSystemProperties to @DurationUnit(ChronoUnit.SECONDS) private Duration sessionTimeout ? Duration.ofSeconds(30); private Duration</String,> </String,> </String,> </String,> • Duration.deMillis(1000); Public Duration getSessionTimeout() a return this.sessionTimeout; ? public void setSessionTimeout(Duration sessionTimeout) to this.sessionTimeout ? sessionTimeout; ? public Duration getReadTimeout() ? return this.readTimeout; ? public void setReadTimeout(Duration readTimeout) ? To specify a session timeout of 30 seconds, 30, PT30S, and 30s are all equivalent. A read timeout of 500 ms can be specified in any of the following ways: 500, PT0.5S, and 500ms. You can also use any of the supported drives. These are: ns for nanoseconds for microseconds ms for milliseconds s for seconds m for minutes h for hours d for days The default unit is milliseconds and can be overridden using @DurationUnit as illustrated in the previous sample. Note that the @DurationUnit only supported with JavaBean-style property binding using getters and setters. Not supported with constructor binding. If you are updating a Long property, be sure to set the unit (@DurationUnit) if it is not milliseconds. Doing so provides a transparent update path and supports a much more complete format. In addition to durations, Spring Boot can also work with the java.time.Period type. The following formats can be used in application properties: a regular int representation (using days as the default unit unless a @PeriodUnit) The standard ISO-8601 format used by java.time.Period A simpler format where pairs of values and units are docked (for example. 1y3d means 1 year and 3 days) The following units support the simple format: and for m years for months w for weeks d for days The java.time.Period type never actually stores the number of weeks, it is a shortcut that means 7 days. Spring Framework has a DataSize value type that expresses a size in bytes. If you expose a DataSize property, the following formats are available in application properties: a normal long representation (using bytes as the default unit unless a @DataSizeUnit) A more readable format where the value and drive are docked (for example. 10MB means 10 megabytes) Consider the following example: @ConfigurationProperties(app.io) public class AppIoProperties a @DataSizeUnit(DataUnit.MEGABYTES) private DataSize bufferSize ? DataSize.ofMegabytes(2); private DataSize sizeThreshold ? DataSize.ofBytes(512); public DataSize getBufferSize() á return this.bufferSize; • public void setBufferSize(DataSize bufferSize) to this.bufferSize - bufferSize; • Public DataSize GetSizeThreshold() to return this.sizeThreshold; ? public void setSizeThreshold(DataSize sizeThreshold) - this.sizeThreshold ? sizeThreshold; To specify a buffer size of 10 megabytes, 10 and 10 MB are A size threshold of 256 bytes can be specified as 256 or 256B. You can also use any of the supported drives. These are: B for KB bytes for kilobytes MB for megabytes GB for gigabytes TB for terabytes The default unit is bytes and can be overridden by using as illustrated in the previous sample. If you are updating a Long property, be sure to set the unit (@DataSizeUnit) if they are not bytes. Doing so provides a transparent update path and supports a much more complete format. Spring Boot attempts @ConfigurationProperties validate classes as long as they are annotated with spring @Validated annotation. You can use JSR-303 javax.validation constraint annotations directly in the configuration class. To do this, ensure that a supported implementation of JSR-303 is in the class path, and then add constraint annotations to the fields, as shown in the following example: @ConfigurationProperties(prefix-acme) @Validated public class AcmeProperties - @NotNull private InetAddress remoteAddress; // ... getters and setters: You can also trigger validation by annotating @Bean method that creates configuration properties with @Validated. To ensure that validation is always triggered for nested properties, even when no properties are found, the associated field must be annotated with @Valid. The following example is based on the previous AcmeProperties example: @ConfigurationProperties(prefix-acme) @Validated acmeProperties public class - @NotNull inetAddress remoteAddress; @Valid Security Security - new Security(); // ... getters and set public static class Security @NotEmpty a public user name String; // ... Getters and setters: You can also add a custom spring validator by creating a bean definition named configurationPropertiesValidator. The method @Bean be declared static. The configuration property validator is created very early in the application lifecycle, and declaring the @Bean method as static allows you to create the bean without having to instantiate the @Configuration. Doing so avoids any issues that may be caused by early instantiating. The spring-boot-actuator module includes an endpoint that exposes all @ConfigurationProperties bean. Point your web browser to /actuator/configprops or use the equivalent JMX endpoint. See the Production-ready features section for more information. The annotation @Value is a primary wrapper feature and does not provide the same features as type-safe configuration properties. The following table summarizes the features supported by @ConfigurationProperties and @Value: If you define a set of configuration keys for your own components, we recommend that you group them into a POJO annotated with @ConfigurationProperties. Doing so will provide you with a structured and safe object for the type you can inject into your own beans. If you would like @Value use your information, we recommend that you refer to property names using their canonical form (case of kebab using only lowercase letters). This will allow Spring Boot to use the same logic as when relaxing the @ConfigurationProperties. For example, @Value of the demo.item-price- data list property) will collect the demo.item-price and demo.itemPrice forms from the application.properties file, as well as system environment. Instead, if you used @Value(-demo.itemPrice), the demo price and the DEMO_ITEMPRICE. Spring profiles provide a way to segregate parts of your application configuration and make it available only in certain environments. Any @Component, @Configuration, or @ConfigurationProperties can be marked with @Profile to limit when loaded, as shown in the following example: @Configuration(proxyBeanMethods ? false) @Profile(production) public class ProductionConfiguration to // ... If @ConfigurationProperties beans are registered through @EnableConfigurationProperties instead of auto-scan, the @Profile annotation must be specified in the @Configuration class that has the @EnableConfigurationProperties. In the event that the @ConfigurationProperties are scanned, you can specify @Profile in the class itself @ConfigurationProperties. You can use a spring.profiles.active Environment property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in application.properties, as shown in the following example: spring.profiles.active-dev,hsqldb You can also specify it on the command line by using the following switch: --spring.profiles.active-dev,hsqldb. The spring.profiles.active property follows the same sort rules as other properties: the highest PropertySource wins. This means that you can specify active profiles in application.properties and then replace them using the command-line switch. Sometimes it's helpful to have profile-specific properties that are added to active profiles instead of replacing 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 (that is, in addition to those enabled by the spring.profiles.active property). See the setAdditionalProfiles() method in SpringApplication. For example, when an application with the following properties is run using the switch, --spring.profiles.active-prod, the proddb and prodmq profiles are also activated: --- my.property: fromyamlfile --- spring.profiles: prod spring.profiles.include: - proddb - prodmq You can set active profiles programmatically by calling SpringApplication.setAdditionalProfiles(...) before the application runs. It is also possible to activate profiles using Spring's ConfigurableEnvironment interface. Specific variants of the application.properties profile (or application.yml) and files referenced through @ConfigurationProperties are considered files and are loaded. See Profile-Specific Properties for more information. Spring uses the Commons record for the entire internal record, but leaves the underlying registry implementation open. Default settings are provided for Java Util Logging, Log4J2, and Logback. In each case, the loggers are preconfigured to use the console output with output is also available. By default, if you use Starters, Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J work correctly. There are a lot of logging frameworks available for Java. Don't worry if the list above seems confusing. Typically, you do not need to change the registry dependencies and Spring Boot defaults work correctly. When you deploy the application to a servlet container or application server, logging through the Java Util registration API is not routed to application logs. This prevents logging

performed by the container or other applications that have been deployed to it from appearing in the application logs. Spring Boot's default log output is similar to the following example: 2019-03-05 10:57:51.112 INFO 45469 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat /7.0.52 2019-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat]. [localhost]. [/] : Initialization of Spring Embedded WebApplicationContext 2019-03-05 10:57:51.253 INFO 45469 --- [ost-startStop-1] o.s.web.context.context.context.ContextLoader : Root WebApplicationContext: initialization completed at 1358 ms 2019-03-05 10:57:51.698 INFO 45469 --- [ost-startStop-11] o.s.b.c.e.ServletRegistrationBean : Allocation Servlet: 'dispatcherServlet' to [/] 2019-03-05 10:57:51.702 INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [/*] The following elements are generated: Date and time: millisecond accuracy and easily sortable. Logging level: ERROR, WARN, INFO, DEBUG or TRACE. Process ID. A message --- to distinguish the start of actual log messages. Thread name: Enclosed in square brackets (can be truncated for console output). Registrar name: This is usually the name of the source class (often abbreviated). The log message. Logback does not have a FATAL level. Maps to ERROR. The default logging setting echoes messages in 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 the application with a --debug flag. $ java -jar myapp.jar --debug You can also specify debug-true in your application.properties. When debug mode is enabled, a selection of primary loggers (embedded container, Hibernate, and Spring Boot) is configured to generate more information. Enabling debug mode does not configure the application to log all messages at the DEBUG level. Alternatively, you can enable a tracking mode by starting the with a --trace flag (or trace-true in your application.properties). This allows trace logging for a selection of primary loggers (embedded container, Hibernate schema generation, and the entire spring portfolio). If the terminal supports ANSI, color output is used for readability. You can set spring.output.ansi.enabled spring.output.ansi.enabled a supported value to override autodiscover. Color encoding is configured using the conversion word %clr. In its simplest form, the converter colors the output according to the logging level, as shown in the following example: The following table describes assigning the logging levels to the colors: FATAL Level Color Red ERROR Red WARN GREEN YELLOW DEBUG Green TRACE Green Alternatively, you can specify the color or style to use by providing it as an option for conversion. For example, to make the text yellow, use the following settings: %clr(%d-yyyy-MM-dd HH:mm:ss. SSS-yellow: The following colors and styles are supported: blue cyan blue blue blue black magenta magenta by default, Spring Boot logs only to the console, and does not write log files. If you want to write log files in addition to console output, you must set a logging.file.name or logging.file.path property (for example, in application.properties). The following table shows how logging.*: Table 7 properties can be used together. The log properties logging.file.name logging.file.path Description example (none) (none) Console log only. 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 rotate when they reach 10 MB, and as with console output, ERROR, WARN, and info level messages are logged by default. Size limits can be changed using the logging.file.max-size property. Rotated log files from the last 7 days are retained by default unless the logging.file.max-history property is set. The total size of log files can be limited by using logging.file.total-size-cap. When the total size of the log files exceeds that threshold, the backups will be deleted. To force log archiving to be cleaned up when you start the application, use the logging.file.clean-history-on-start property. Log 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. All supported logging systems can have logger levels set to spring Environment (for example, in application.properties) using logging.level. <logger-name>•<level> where the level is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL or OFF. The root logger can be configured using logging.level.root. In the The following shows the potential logging settings in application.properties: logging.level.root-warn logging.level.org.springframework.web-debug logging.level.org.hibernate-error It is also possible to set logging levels using environment variables. For example, LOGGING_LEVEL_ORG_SPRINGFRAMEWORK_WEB-DEBUG will set org.springframework.web to DEBUG. The above approach will only work for package-level logging. Because relaxed binding always converts environment variables</level> </logger-name> </logger-name> lowercase, it is not possible to configure logging for an individual class in this way. If you need to configure logging for a class, you can use the SPRING_APPLICATION_JSON. It is often useful to be able to group related loggers so that everyone can be configured at the same time. For example, you might typically change logging levels for all Tomcat-related loggers, but you cannot easily remember top-level packages. To help with this, Spring Boot allows you to define log groups in your spring environment. For example, the following shows how you can define a tomcat group by adding it to your application.properties: logging.group.tomcat-org.apache.catalina, org.apache.coyote, org.apache.tomcat Once defined, you can change the level of all loggers in the group with a single line: logging.level.tomcat-TRACE Spring Boot includes the following predefined log groups that can be used from the factory: Name Loggers web org.springframework.core.codec, org.springframework.http, org.springframework.web, org.springframework.boot.actuate.endpoint.web, org.springframework.boot.web.servlet.ServletContextInitializerBeans sql org.springframework.jdbc.core, org.hibernate.SQL , org .jooq.tools.LoggerListener Different logging systems can be activated by including the appropriate libraries in the class path and can be further customized by providing an appropriate configuration file at the root of the class path or at a location specified by the following Spring Environment property: logging.config. You can force Spring Boot to use a particular logging system by using the org.springframework.boot.logging.LoggingSystem system property. The value must be the fully qualified class name of a LoggingSystem implementation. You can also completely disable Spring Boot logging settings by using a value of none. Because the registry is initialized before you create ApplicationContext, it is not possible to control the @PropertySources in Spring @Configuration. The only way to change the logging system or disable it completely is through the system properties. Depending on your registration system, The following files are loaded: Logging System Customization Logback-spring.xml, logback-spring.groovy, logback.xml, or logback.groovy Log4j2 log4j2-spring.xml or log4j2.xml JDK (Java Util Logging) logging.properties When possible, we recommend that you use the -spring variants for logging configuration (for example, logback-spring.xml instead of logback.xml). If you use standard configuration locations, Spring cannot fully control registry initialization. There are known class loading issues with Java Util Logging that cause problems when running from an 'executable jar'. We recommend that you avoid it when running from 'jar executable' if possible. To help with customization, some other properties are transferred from the spring environment to system properties, as described in the following table: Spring Environment System Property Property logging.exception-conversion-word LOG_EXCEPTION_CONVERSION_WORD The conversion word used when logging exceptions. logging.file.clean-history-on-start LOG_FILE_CLEAN_HISTORY_ON_START If you want to clean the archive log files at startup (if LOG_FILE enabled). (Supported only with the default Logback settings.) logging.file.name LOG_FILE If defined, is used in the default logging settings. logging.file.max-size LOG_FILE_MAX_SIZE Maximum log file size (if LOG_FILE enabled). (Supported only with the default Logback settings.) logging.file.max-history LOG_FILE_MAX_HISTORY Maximum number of archive log files to maintain (if LOG_FILE enabled). (Supported only with the default Logback settings.) logging.file.path LOG_PATH If defined, is used in the default logging settings. logging.file.total-size-cap LOG_FILE_TOTAL_SIZE_CAP The Total size of log backups to keep (if LOG_FILE enabled). (Supported only with the default Logback settings.) logging.pattern.console CONSOLE_LOG_PATTERN the logging pattern to use in the console (stdout). (Supported only with the default Logback settings.) logging.pattern.dateformat LOG_DATEFORMAT_PATTERN Appender pattern for the registration date format. (Supported only with the default Logback settings.) logging.pattern.file FILE_LOG_PATTERN The logging pattern to use in a file (if LOG_FILE is enabled). (Supported only with the default Logback settings.) logging.pattern.level LOG_LEVEL_PATTERN the format to use when rendering the log level (default %5p). (Supported only with the default Logback settings.) logging.pattern.rolling-file-name ROLLING_FILE_NAME_PATTERN pattern for rolled-over log file names (default $. LOG_FILE.%d-yyyy-MM-dd-.%i.gz). (Supported only with the default Logback settings.) PID PID The current process ID (discovered if possible and when not yet defined as an operating system environment variable). All supported logging systems can query system properties when analyzing their configuration files. See the default settings in spring-boot.jar for examples: Java Util Log4j 2 If you want to use a placeholder in a log property, you must use Spring Boot syntax and not the underlying frame syntax. In particular, if you use Logback, you must use : as a delimiter between a property name and its default value and not use :-. You can add MDC and other ad hoc content to log lines by replacing only the LOG_LEVEL_PATTERN (or logging.pattern.level with Logback). For example, if you use logging.pattern.level-user:%X-user- %5p, the default log contains an MDC entry for user, if any, as shown in the following example. 2019-08-30 12:30:04.031 user:someone INFO 22174 --- [nio-8080-exec-0] demo. Spring Boot Authenticated Request Management Driver includes a number of Logback extensions that can help with advanced configuration. You can use these extensions in the logback-spring.xml configuration file. Because the standard logback.xml configuration file loads too soon, you cannot use extensions in it. You need use logback-spring.xml or set a logging.config property. Extensions cannot be used with Logback configuration analysis. If you try to do so, making changes to the configuration file results in an error similar to one of the following: ERROR in :71 - no applicable action for [springProperty], Current ElementPath is [[configuration][springProperty]] ERROR in :71 - no applicable action for [springProfile], current ElementPath is [[configuration][springProfile]] The <springProfile>tag allows you to optionally include or exclude configuration sections based on the active Spring. Profile sections are supported anywhere within the <configuration>element. Use the name attribute to specify which profile accepts the configuration. The <springProfile>tag can contain a profile name (for example, essay) or a profile expression. A profile expression allows you to express more complicated profile logic, for example, production &(eu-central - eu-west). Refer to the reference guide for more details. The following list <springProfile name-staging> <!-- configuration to be enabled when the staging profile is active --> </springProfile> shows three sample profiles: <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> the <springProperty>tag allows you to expose properties of the spring environment for use in Logback. Doing so can be useful if you want to access the values from the application.properties file in the Logback configuration. The tag works similarly to the standard <property>Logback tag. However, instead of specifying a direct value, specify the source of the property (from the environment). If you need to store the property elsewhere other than in local scope, you can use the scope attribute. If you need a reservation value (in case the property is not set in your environment), you can use the defaultValue attribute. The following example shows how to expose the properties for use in Logback: <springProperty scope-context name-fluentHost source-myapp.fluentd.host defaultvalue-localhost></springProperty> <appender name-FLUENT class-ch.qos.logback.more.appenders.DataFluentAppender> <remoteHost>$.</remoteHost> </appender> The source must be specified in the case of kebab (such as my.property-name). However, properties can be added to the environment using relaxed rules. Spring Boot supports localized messages so that the application can serve users of different language preferences. So Spring Boot looks for the presence of a message resource packet at the root of the class path. Automatic configuration applies when the default properties file for the configured resource package is available (that is, messages.properties by</property> </springProperty> </springProfile> </configuration> </springProfile> </springProfile> If the resource pack contains only language-specific property files, you must add the default value. If no property file matching any of the configured base names is found, there will be no MessageSource configured automatically. The base name of the resource package, as well as several other attributes, can be configured using the spring.messages namespace, as shown in the following example: spring.messages.basename-messages,config.i18n.messages spring.messages.fallback-to-system-locale-false spring.messages.basename supports a comma-separated list of locations, either a package qualifier or a resource resolved from the root of the class path. See MessageSourceProperties for more supported options. Spring Boot provides integration with three JSON mapping libraries: Jackson is the preferred and default library. Automatic configuration is provided for Jackson and Jackson is part of spring-boot-starter-json. When Jackson is in the class path, an ObjectMapper bean is automatically configured. Several configuration properties are provided to customize ObjectMapper settings. Automatic configuration is provided for Gson. When Gson is in the class path, a Gson bean is automatically configured. Several spring.gson.* configuration properties are provided to customize the settings. To take more control, you can use one or more GsonBuilderCustomizer beans. Automatic configuration is provided for JSON-B. When the JSON-B API and an implementation are in the class path, a Jsonb bean is automatically configured. The preferred JSON-B implementation is Apache Johnzon for which dependency management is provided. Spring Boot is suitable for web application development. You can create a stand-alone HTTP server using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the spring-boot-starter-web module to get up and running quickly. You can also create reactive web applications using the spring-boot-starter-webflux module. If you haven't developed a Spring Boot web app yet, you can follow the Hello World! in the Getting Started section. The Spring Web MVC Framework (often called Spring MVC) is a rich model view controller web framework. Spring MVC allows you @Controller create special @RestController or beans to handle incoming HTTP requests. Handler methods map to HTTP using annotations @RequestMapping. The following code shows a typical @RestController that serves JSON data: @RestController @RequestMapping(value-/users) public class MyRestController a @RequestMapping(value-/-user-, method-RequestMethod.GET) public User getUser(@PathVariable Long user) to // ... ? @RequestMapping(value?/user-/customers, List<Customer> getUserCustomers(@PathVariable Long user) ? // @PathVariable @RequestMapping ... Spring MVC es parte del marco de primavera principal, y </Customer> </Customer> information is available in the reference documentation. There are also several guides covering Spring MVC available in spring.io/guides. Spring Boot provides automatic configuration for Spring MVC that works well with most applications. Automatic configuration adds the following features in addition to Spring defaults: Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans. Support for the static resource service, including webjars support (discussed later in this document)). Automatic registration of Beans Converter, GenericConverter and Formatter. Support for HttpMessageConverters (covered later in this document). Automatic messagecodesResolver registration (covered later in this document). Support for static index.html. Favicon custom support (covered later in this document). Automatic use of a ConfigurableWebBindingInitializer bean (covered later in this document). If you want to maintain those Spring Boot MVC customizations and perform more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own @Configuration class of type WebMvcConfigurer but @EnableWebMvc. If you want to provide custom instances of RequestMappingHandlerMapping, RequestMappingHandlerAdapter, or ExceptionHandlerExceptionResolver and preserve Spring Boot MVC customizations, you can declare a bean of type WebMvcRegistrations and use it to provide custom instances of those components. If you want to take full control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc or alternatively add your own delegatingWebMvcConfiguration @Configuration annotated as described in the @EnableWebMvc Javadoc. Spring MVC uses the HttpMessageConverter interface to convert HTTP requests and responses. Sensitive defaults are included at the factory. For example, objects can be automatically converted to JSON (using the Jackson library) or XML (using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in UTF-8. If you need to add or customize converters, you can use the Spring Boot HttpMessageConverters class, as shown in the following list: import org.springframework.boot.autoconfigure.http.HttpMessageConverters; import org.springframework.context.annotation.*; import org.springframework.http.converter.*; @Configuration(proxyBeanMethods ? false) of the public class MyConfiguration - @Bean public HttpMessageConverters customConverters() - HttpMessageConverter<?> additional ... HttpMessageConverter<?> another ... return new HttpMessageConverters(additional, another); • Any HttpMessageConverter bean that is present the context is added to the converter list. You can also override the default converters in the same way. If you use Jackson to serialize and deserialize JSON data, you may want to write your own JsonSerializer and JsonDeserializer classes. Custom serializers typically register with through a module, but Spring Boot provides an alternative @JsonComponent annotation that makes it easy to register Spring Beans directly. You can use annotation @JsonComponent directly in JsonSerializer, JsonDeserializer, or KeyDeserializer implementations. You can also use it in classes that contain serializers/deserializers as internal classes, as shown in the following example: import java.io.*; import com.fasterxml.jackson.core.*; import com.fasterxml.jackson.databind.*; import org.springframework.boot.jackson.*; @JsonComponent public class Example to public static class Serializer extends JsonSerializer<SomeObject> to // ... Deserializer public static class extends JsonDeserializer<SomeObject> ? // ... All be @JsonComponent in ApplicationContext are automatically registered with Jackson. Because @JsonComponent metanota is @Component, the usual component analysis rules apply. Spring MVC has a strategy for generating error codes to represent error messages from binding errors: MessageCodesResolver. If you set the spring.mvc.message-codes-resolver-format PREFIX_ERROR_CODE or POSTFIX_ERROR_CODE property, Spring Boot creates one for you (see the enumeration in DefaultMessageCodesResolver.Format). By default, Spring Boot serves static content from a directory named /static (or /public or /resources or /META-INF/resources) in the class path or from the ServletContext root. Use Spring MVC ResourceHttpRequestHandler so that you can modify that behavior by adding your own WebMvcConfigurer and overriding the addResourceHandlers method. In a stand-alone web application, the container's default servlet is also enabled and acts as a reservation, serving content from the ServletContext root if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through DispatcherServlet. By default, resources are allocated in /**, but you can adjust it with the spring.mvc.static-path-pattern property. For example, relocating all resources to /resources/** can be accomplished as follows: spring.mvc.static-path-pattern-/resources/** You can also customize static resource locations by using the spring.resources.static-locations property (replacing the default values with a list of directory locations). The context path of the root Servlet, /, is also automatically added as a location. In addition to the standard static resource locations mentioned above, a special case is created for Webjars content. All resources with a path in /webjars/** are served from jar files if they are packaged in the Webjars format. Do not use the src/main/webapp directory if your application is like a jar. Although this directory is a common standard, it only works with war packaging, and most build tools silently ignore it if it generates a jar. Spring Boot also supports advanced resource management features</SomeObject> </SomeObject> </SomeObject> by Spring MVC, allowing use cases such as dedocaying static resources cache or using version-independent URLs for Webjars. To use version agnostic URLs for Webjars, add the webjars-locator-core dependency. Then declare your Webjar. Using jQuery as an example, adding /webjars/jquery/jquery.min.js results in /webjars/jquery/x.y.z/jquery.min.js where x.y.z is the Webjar version. If you use JBoss, you must declare the webjars-locator-jboss-vfs dependency instead of the webjars-locator-core. Otherwise, all Webjars resolve to a 404. To use cache decommunication, the following configuration configures a cache dedob solution for all static resources, effectively adding a content hash, such as <link href-/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css> , in the URLs: spring.resources.chain.strategy.resources./** Links to resources are rewritten into runtime templates, thanks to ResourceUrlEncodingFilter that is automatically configured for Thymeleaf and FreeMarker. You must manually declare this filter when using JSP. Currently, other template engines are not automatically supported, but can be with custom macros/template helpers and the use of ResourceUrlProvider. When loading resources dynamically with, for example, a JavaScript module loader, re-name files is not an option. That's why other strategies are also supported and can be combined. A fixed strategy adds a static version string to the URL without re-resing the file name, as shown in the following example: 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.strategy.fixed.. JavaScript modules located in /js/lib/ use a fixed versioning strategy (/v12/js/lib/mymodule.js), while other resources continue to use content one (<link href-/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css>). See ResourceProperties for more supported options. Spring Boot supports static and templated welcome pages. First look for an index.html file in the configured static content locations. If one is not found, look for an index template. If any of them are found, it is automatically used as the app's welcome page. As with other static resources, Spring Boot searches for a favicon.ico in the configured static content locations. If such a file is present, it is automatically used as the favicon of the application. Spring MVC can assign incoming HTTP requests to controllers by examining the request path and matching the mappings defined in the application (for example, annotations @GetMapping controller methods). Spring Boot chooses to disable suffix pattern matching by default, which means that requests such as GET /projects/spring-boot.json will not match @GetMapping(/projects/spring-boot) mappings. This is considered a best practice for Spring MVC applications. This feature was primarily useful in the past for HTTP clients that did not send appropriate ACCEPT request headers; we needed to make sure we sent the right type of content to the customer. Today, content negotiation is much more reliable. There are other ways to deal with HTTP clients that do not constantly send appropriate OK request headers. Instead of using suffix matching, we can use a query parameter to make sure that requests such as GET /projects/spring-boot?format-json are assigned to @GetMapping(/projects/spring-boot): spring.mvc.contentnegotiation.favor-parameter-true - We can rename the parameter, Which is format by default: - spring.mvc.contentnegotiation.parameter-name-myparam - We can also register additional file extensions/media types with: spring.mvc.contentnegotiation.media-types.markdown-text/markdown Suffix pattern matching is deprecated and will be removed in a future release. If you understand the warnings and want your application to use suffix pattern matching, the following configuration is required: spring.mvc.contentnegotiation.favor-path-extension-true spring.mvc.pathmatch.use-suffix-pattern-true Alternatively, Instead of opening all suffix patterns, it is safer to support only registered suffix patterns: spring.mvc.contentnegotiation.favor-path-extension-true spring.mvc.pathmatch.use-registered-suffix-pattern-true - You can also register additional file extensions/media types with: . If you create your own @Bean ConfigurableWebBindingInitializer, Spring Boot automatically configures Spring MVC to use it. In addition to REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of template technologies, including Thymeleaf, FreeMarker, and JSP. In addition, many other template engines include their own Spring MVC integrations. Spring Boot includes automatic configuration support for the following template engines: FreeMarker Groovy Thymeleaf Mustache If possible, JSP should be avoided. There are several known limitations when using them with embedded servlet containers. When you use one of these template engines with the default settings, templates are automatically collected from src/main/resources/templates. Depending on how you run the application, IntelliJ IDEA sorts the class path differently. Running the application in the IDE from its main method results in a different order than when the application is run using Maven or Gradle or from its packaged jar. This can cause Spring Boot to not be able to find the templates in the class path. If you have this problem, you can reorder the class path in the IDE to first place the classes and resources of the Alternatively, you can configure the template prefix to search each template directory class path, as follows: classpath*:/templates/. By default, Spring Boot provides an /error mapping that handles all errors in a sensible way and registers as a global error page in the servlet container. For machine clients, it generates a JSON response with details of the error, HTTP status, and exception message. For browser clients, there is a whitelabel error view that represents the same data in HTML format (to customize it, add a view that resolves the error). To override the default behavior completely, you can implement ErrorController and register such a bean definition or add an ErrorAttributes bean to use the existing mechanism but replace the content. The BasicErrorController can be used as a base class for a Custom ErrorController. This is especially useful if you want to add a handler for a new content type (the default is to handle text/html specifically and provide a reservation for everything else). To do this, extend BasicErrorController, add a public method with a @RequestMapping that has a produces attribute, and create a bean of the new type. You can also define a class annotated with @ControllerAdvice to customize the JSON document to return for a particular handler or exception type, as shown in the following example: @ControllerAdvice(basePackageClasses ? AcmeController.class) Public Class AcmeControllerAdvice Extends ResponseEntityExceptionHandler - @ExceptionHandler(YourException.class) @ResponseBody ResponseEntity <?? > handleControllerException(HttpServletRequest, request request, The value of HttpStatus can be returned to the httpStatus(request) property; returns a new Detortion Value <> (new CustomErrorType(status.value(), ex.getMessage()), status); á private HttpStatus getStatus(HttpServletRequest request) to Integer statusCode ? (Integer) request.getAttribute(javax.servlet.error.status_codeHttpStatus.INTERNAL_SERVER_ERROR); In the example above, if YourException is thrown by a handler defined in the same package as AcmeController, a JSON representation of the CustomErrorType POJO is used instead of the ErrorAttributes representation. If you want to display a custom HTML error page for a particular status code, you can add a file to an /error directory. Error pages can be static HTML (that is, added in any of the static resource directories) or compiled using templates. The file name must be the exact status code or a serial mask. For example, to map 404 to a static HTML file, the directory structure would be as follows: src/ +- main/ +- java/ + <source code->+- resources/ +- public/ +- error/ á +- 404.html +- <other-assets->To all 5xx errors by using a FreeMarker template, the directory structure would be as follows: src/ +- main/ +- java/ á + <source code->+- resources/ +- templates/ +- error/ +- 5xx.ftlh +- <other templates->For more complexes, you can also add beans that implement the</other> </source> </other> </source> </source> interface, as shown in the following example: The public class MyErrorViewResolver implements ErrorViewResolver to public @Override ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String, object-> model) ? Use the request or status to optionally return a ModelAndView return ... For applications that do not use Spring MVC, you can use the ErrorPageRegistrate interface to directly register ErrorPages. This abstraction works directly with the underlying embedded servlet container and works even if you don't have a Spring MVC DispatcherServlet. @Bean Public ErrorPageRegister errorPageRegistrate()- return new MyErrorPageRegistrar(); ? // ... The private static class MyErrorPageRegistrar implements ErrorPageRegister @Override public void registerErrorPages(ErrorPageRegistry registry) to registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, /400)); If you register an ErrorPage with a path that ends up being handled by a filter (as is common with some non-Spring web frameworks, such as Jersey and Wicket), the filter has to be explicitly registered as an ERROREs distributor, as shown in the following example: @Bean Public Record FilterRegistrationBean myFilter() ? FilterRegistrationBean registration ? registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class)); return record; • Note that the default FilterRegistrationBean does not include the ERROR distributor type. When deployed to a servlet container, Spring Boot uses its error page filter to forward an error-state request to the appropriate error page. This is necessary because the Servlet specification does not provide an API for registering error pages. Depending on the container in which you are deploying the war file and the technologies that your application uses, some additional configuration may be required. The error page filter can only forward the request to the correct error page if the response has not yet been committed. By default, WebSphere Application Server 8.0 and later commit the response after the service method of a servlet completes successfully. You must disable this behavior by setting com.ibm.ws.webcontainer.invokeFlushAfterService to false. If you are using Spring Security and want to access the principal on an error page, you must configure the Spring Security filter to be invoicing in error submissions. To do this, set the spring.security.filter.dispatcher-types property to async, error, forward, request. If you develop a RESTful API that makes use of hypermedia, Spring Boot provides automatic configuration for Spring HATEOAS that works well with most applications. Automatic configuration replaces the need to use @EnableHypermediaSupport and records a to facilitate the creation of hypermedia-based applications, including LinkDiscoverers (for client-side support) and an ObjectMapper configured to correctly calculate responses in desired responses </String,> </String,> ObjectMapper is customized by setting the various spring.jackson.* properties or, if any, by a Jackson2ObjectMapperBuilder bean. You can take control of Spring HATEOAS settings using @EnableHypermediaSupport. Note that doing so disables the ObjectMapper customization described above. @Configuration(proxyBeanMethods ? false) public class MyConfiguration a @Bean public WebMvcConfigurer corsConfigurer() - return new WebMvcConfigurer() @Override ? Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the Servlet API, is fully asynchronous and non-blocking, and implements the specification of reactive sequences through the Reactor project. Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based is quite close to the Spring MVC model, as shown in the following example: public class @RestController @RequestMapping(/users) MyRestController to @GetMapping(/-user-) public Mono<User> getUser(@PathVariable Long user) to // ... a @GetMapping(/-user-/customers) public Flux<Customer> getUserCustomers(@PathVariable Long user) á // @PathVariable @DeleteMapping ... <User> WebFlux.fn, the functional variant, separates the routing configuration from the actual control of requests, as shown in the following example: @Configuration(proxyBe Public Class RoutingConfiguration of public class @Bean public RouterFunction<ServerResponse> monoRouterFunction(UserHandler userHandler) to return route(GET(/-user-APPLICATION_JSON APPLICATION_JSON). Route(DELETE(/-user-). y(accept(APPLICATION_JSON)), userHandler::d eleteUser); ? ? @Component public class UserHandler to public Mono<ServerResponse> getUser(ServerRequest request) ? // ... <ServerResponse> <ServerResponse> You can define as many RouterFunction beans as you want to modulate the router definition. Beans can be ordered if you need to apply a precedence. To get started, add the spring-boot-starter-webflux module to your application. Adding spring-boot-starter-web and spring-boot-starter-webflux modules to your application results in Spring Boot automatically configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add spring-boot-starter-webflux to their Spring MVC application to use the reactive WebClient. You can still apply your choice by setting the chosen application type to SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE). Spring Boot provides automatic configuration for Spring WebFlux that works well with most applications. </ServerResponse></ServerResponse></ServerResponse></ServerResponse></User></Customer></User>Automatic configuration adds the following features in addition to Spring's default values: If you want to maintain Spring Boot WebFlux features and want to add additional WebFlux settings, you can add your own @Configuration class of type WebFluxConfigurer but no @EnableWebFlux. If you want to take full control of Spring WebFlux, you can add your own @Configuration annotated with @EnableWebFlux. Spring WebFlux uses the HttpMessageReader and HttpMessageWriter interfaces to convert HTTP requests and responses. They are configured with CodecConfigurer to have reasonable default values by looking at the libraries available in the class path. Spring Boot provides dedicated configuration properties for codecs, spring.codec.*. It also applies greater customization by using CodecCustomizer instances. For example, spring.jackson.* configuration keys apply to the Jackson codec. If you need to add or customize codecs, you can create a custom CodecCustomizer component, as shown in the following example: import org.springframework.boot.web.codec.CodecCustomizer; @Configuration(proxyBeanMethods - false) public class MyConfiguration - @Bean public CodecCustomizer myCodecCustomizer() - return codecConfigurer -> ? // ... You can also take advantage of Boot custom JSON serializers and deserializers. By default, Spring Boot serves static content from a directory named /static (or /public or /resources or /META-INF/resources) in the class path. Use Spring WebFlux ResourceWebHandler so that you can modify that behavior by adding your own WebFluxConfigurer and overriding the addResourceHandlers method. By default, resources are allocated to /**, but you can adjust it by setting the spring.webflux.static-path-pattern property. For example, relocating all resources to /resources/** can be accomplished as follows: spring.webflux.static-path-pattern-/resources/** You can also customize static resource locations using spring.resources.static-locations. Doing so replaces the default values with a list of directory locations. If you do, the default welcome page discovery changes to your custom locations. Therefore, if there is an index.html in any of its locations at startup, it is the main page of the application. In addition to the standard static resource locations listed above, a special case is created for Webjars content. All resources with a path in /webjars/** are served from jar files if they are packaged in the Webjars format. Spring WebFlux applications do not rely strictly on the Servlet API, so they cannot be deployed as war files and do not use the src/main/webapp directory. Spring Boot supports static and templated. First look for an index.html file in the configured static content locations. If one is not found, look for an index template. If any of them are found, it is automatically used as the app's welcome page. In addition to REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Dynamic. WebFlux supports a variety of template technologies, including Thymeleaf, FreeMarker and Mustache. Spring Boot includes automatic configuration support for the following template engines: FreeMarker Thymeleaf Mustache When you use one of these template engines with the default settings, your templates are automatically collected from src/main/resources/templates. Spring Boot provides a WebExceptionHandler that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered latter. For machine clients, it generates a JSON response with details of the error, HTTP status, and exception message. For browser clients, there is a whitelabel error handler that represents the same data in HTML format. You can also provide your own HTML templates for displaying errors (see next section). The first step in customizing this feature often involves using the existing mechanism, but replacing or increasing the content of the error. To do this, you can add a bean of type ErrorAttributes. To change the error handling behavior, you can implement ErrorWebExceptionHandler and register such a bean definition. Because a WebExceptionHandler is fairly low-level, Spring Boot also provides a handy AbstractErrorWebExceptionHandler to allow you to handle errors in a functional way in WebFlux, as shown in the following example: The CustomErrorWebExceptionHandler public class extends AbstractErrorWebExceptionHandler ? Defines constructor here @Override protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes errorAttributes) - return RouterFunctions .route(aPredicate, aPredicate, .yRoute(anotherPredicate, anotherHandler); For a more complete image, you can also subclass DefaultErrorWebExceptionHandler directly and override specific methods. If you want to display a custom HTML error page for a particular status code, you can add a file to an /error directory. Error pages can be static HTML (that is, added in any of the static resource directories) or created with templates. The file name must be the exact status code or a serial mask. For example, to map 404 to a static HTML file, the directory structure would be as follows: src/ +- main/ +- java/ + <source code->+- resources/ +- public/ +- error/ á +- 404.html +- <other-assets->To assign all 5xx errors by using a mustache template, the directory structure would be as follows: src/ +- main/ +- java/ + <source code->resources/ +- templates/ +- error/ +- 5xx.mustache +- <other templates->Spring WebFlux provides a WebFilter interface that can be to filter HTTP request exchanges of responses. WebFilter beans that are in the application context will be automatically used to filter each interchange. When filter ordering is important, they can implement Ordered or annotated with @Order. Spring Boot auto-configuration can configure web filters automatically. When you do, the</other> </source> </other> </source> </ServerResponse> </ServerResponse> it will be used in the following table: Web Filter Order MetricsWebFilter Ordered.HIGHEST_PRECEDENCE + 1 WebFilterChainProxy (Spring Security) -100 HttpTraceWebFilter Ordered.LOWEST_PRECEDENCE - 10 If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. Jersey and Apache CXF work pretty well at the factory. CXF requires that you register your Servlet or Filter as @Bean in the context of your application. Jersey has some native spring support, so we also provide automatic setup support for it in Spring Boot, along with a starter. To get started with Jersey, include the spring-boot-starter-jersey as a dependency, and then you need a ResourceConfig @Bean in which all endpoints are registered, as shown in the following example: @Component JerseyConfig public class extends ResourceConfig - public JerseyConfig() to register(Endpoint.class); Jersey's support for scanning executable files is quite limited. For example, you cannot search for endpoints in a package that is in a fully executable jar file or web-INF/classes when running an executable war file. To avoid this limitation, you should not use the packages method, and endpoints must be registered individually using the register method, as shown in the previous example. For more advanced customizations, you can also register an arbitrary number of beans that implement ResourceConfigCustomizer. All registered endpoints must be @Components with HTTP resource annotations (@GET and others), as shown in the following example: @Component @Path(/hello) of the Endpoint public class (@GET public String message() - return Hello; ? Because the endpoint is a spring @Component, its lifecycle is managed by Spring and you can use the @Autowired annotation to insert dependencies and use the @Value annotation to insert the external settings. By default, the Jersey servlet is registered and assigned to /*. You can change the mapping by @ApplicationPath to ResourceConfig. By default, Jersey is configured as a Servlet in a @Bean of type ServletRegistrationBean named jerseyServletRegistration. By default, the servlet is initialized lazily, but you can customize that behavior by setting spring.jersey.servlet.load-on-startup. You can disable or override that bean by creating your own bean 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 replace or replace is jerseyFilterRegistration). The filter has a @Order, which you can set with spring.jersey.filter.order. Both the servlet and filter records can be assigned init parameters using spring.jersey.init.* specify a property map. Spring Boot includes support for built-in Tomcat, Jetty, and Undertow servers. Most developers use the appropriate Starter to get a fully configured instance. By default, the embedded server listens for HTTP requests on the port port When using an embedded servlet container, you can register servlets, filters, and all listeners (such as HttpSessionListener) from the De Servlet specification, either by using Spring beans or by searching for Servlet components. Any instance of Servlet, Filter, or *Listener servlet that is a Spring bean is registered in the embedded container. This can be especially convenient if you want to reference a value of application.properties during configuration. By default, if the context contains a single Servlet, it maps to /. For multiple servlet beans, the bean name is used as a route prefix. Filters are assigned to /*. If convention-based mapping is not flexible enough, you can use the ServletRegistrationBean, FilterRegistrationBean, and ServletListenerRegistrationBean classes for full control. It is usually safe to leave the filter beans unordered. If a specific order is required, you must annotate the filter with @Order or have it implement Sorted. You cannot set the order of a filter by annotating its bean method with @Order. If you cannot change the Filter class to add @Order or implement Ordered, you must define a FilterRegistrationBean for Filter and set the order of the record bean using the setOrder(int) method. Avoid setting up a filter that reads the request body Ordered.HIGHEST_PRECEDENCE, as it might go against the application's character encoding settings. If a Servlet filter wraps the request, it must be configured in an order less than or equal to OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER. To view the order of each application filter, enable debug-level logging for the web log group (logging.level.web-debug). Details of registered filters, including their order and URL patterns, will be logged at startup. Use caution when registering filter beans, as they are initialized very early in the application's lifeline. If you need to register a filter that interacts with other beans, consider using a DelegatingFilterProxyRegistrationBean instead. Embedded servlet containers do not directly run the Servlet 3.0+ interface javax.servlet.ServletContainerInitializer or spring's org.springframework.web.WebApplicationInitializer interface. This is an intentional design decision aimed at reducing the risk that third-party libraries designed to run within a war may disrupt Spring Boot applications. If you need to perform servlet context initialization in a Spring Boot application, you must register a bean that implements the org.springframework.boot.web.servlet.ServletContextInitializer interface. The single onStartup method provides access to the ServletContext and, if necessary, can be easily used as an adapter to a WebApplicationInitializer When using an embedded container, automatic registration of classes annotated with @WebServlet, @WebFilter, and @WebListener can be enabled @ServletComponentScan. @ServletComponentScan has no effect on a container, where the container's built-in detection mechanisms are used instead. Under the hood, Spring Boot uses a different type of ApplicationContext for support for embedded servlet containers. ServletWebServerApplicationContext is a special type of WebApplicationContext that is booted by searching for a single ServletWebServerFactory bean. Typically, a TomcatServletWebServerFactory, JettyServletWebServerFactory, or UndertowServletWebServerFactory has been configured automatically. You typically do not need to consider these implementation classes. Most applications are automatically configured and created in their appropriate ApplicationContext and ServletWebServerFactory name. Common servlet container settings can be configured using Spring Environment properties. Typically, you would define properties in the application.properties file. Common server configuration includes: Network configuration: Listen for port for incoming HTTP requests (server.port), interface address to bind to server.address, and so on. Session configuration: If the session is persistent (server.servlet.session.persistent), the session timeout (server.servlet.session.timeout), the location of the session data (server.servlet.session.store-dir), and the session cookie setting (server.servlet.session.cookie.*). Error management: The location of the error page (server.error.path) and so on. SSL HTTP Spring Boot compression attempts to expose common settings, but this is not always possible. For those cases, dedicated namespaces provide server-specific customizations (see server.tomcat and server.undertow). For example, access logs can be configured with features specific to the embedded servlet container. See the ServerProperties class for a complete list. If you need to programmatically configure the embedded servlet container, you can register a Spring bean that implements the WebServerFactoryCustomizer interface. WebServerFactoryCustomizer provides access to configurableServletWebServerFactory, which includes numerous customization setter methods. The following example shows how to programmatically set the port: import org.springframework.boot.web.server.WebServerFactoryCustomizer; import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory; import org.springframework.stereotype.Component; @Component CustomizationBean public class implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> - @Override public void customize(ConfigurableServletWebServerFactory server) to server.setPort(9000); TomcatServletWebServerFactory, JettyServletWebServerFactory, and UndertowServletWebServerFactory are dedicated variants of

ConfigurableServletWebServerFactory that have additional customization setter for Tomcat, Jetty, and Undertow respectively. If the above customization techniques are too limited, you can register the TomcatServletWebServerFactory, JettyServletWebServerFactory, or UndertowServletWebServerFactory bean yourself. </ConfigurableServletWebServerFactory> </ConfigurableServletWebServerFactory> public ConfigurableServletWebServerFactory webServerFactory() to TomcatServletWebServerFactory factory á new TomcatServletWebServerFactory(); 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. Various protected hook methods are also provided in case you need to do something more exotic. See the source code documentation for more information. When you run a Spring Boot application that uses an embedded servlet container (and is packaged as an executable file), there are some limitations to JSP support. With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when started with java -jar, and can also be deployed to any standard container. JSPs are not supported when using an executable jar file. Undertow does not support JSP. Instead, custom error pages should be used. Spring Boot includes support for the following integrated reactive web servers: Netty Reactor, Tomcat, Jetty, and Undertow. Most developers use the appropriate Starter to get a fully configured instance. By default, the embedded server listens for HTTP requests on port 8080. When you automatically configure a Netty or Jetty Reactor server, Spring Boot will create specific beans that will provide HTTP resources to the server instance: ReactorResourceFactory or JettyResourceFactory. By default, those resources will also be shared with the Reactor Netty and Jetty clients for optimal performance, given: the same technology is used for the server and client, the client instance is created using the WebClient.Builder bean automatically configured by Spring Boot Developers can override the resource settings for Jetty and Reactor Netty by providing a custom ReactorResourceFactory or JettyResourceFactory bean - this will apply to both clients and servers. Smart shutdown is compatible with the four integrated web servers (Jetty, Netty Reactor, Tomcat, and Undertow) and Servlet-based, reactive web applications. It occurs as part of closing the context of the application and takes place at the earliest stage of stopping SmartLifecycle beans. This stop processing uses a timeout that provides a grace period during which existing requests will be allowed to complete, but no new requests will be allowed. The exact way no new requests are allowed varies depending on the web server being used. Jetty, Reactor Netty, and Tomcat will no longer accept requests at the network layer. Undertow will accept requests but respond immediately with a non-service response (503). Smart shutdown with Tomcat requires Tomcat 9.0.33 or later. In order to enable successful shutdown, configure the server.shutdown property, as shown in the following example: In order to configure the timeout period, configure the property, as shown in the following example: spring.lifecycle.timeout-per-shutdown-phase-20s Using the correct shutdown with the IDE may not work properly if it does not send an appropriate SIGTERM signal. See your IDE documentation for more details. RSocket is a binary protocol for use in byte stream transports. Enables symmetric interaction models through asynchronous messages that pass over a single connection. The Spring Framework spring-messaging module provides support for RSocket requesters and responders, both on the client and on the server. See the RSocket section of the Spring Framework reference for more details, including an overview of the RSocket protocol. Spring Boot automatically configures an RSocketStrategies bean that provides all the infrastructure needed to encode and decode RSocket payloads. By default, automatic configuration will attempt to configure the following (in order): CBOR codecs with Jackson JSON codecs with Jackson The spring-boot-starter-rsocket starter provides both dependencies. Check out Jackson's support section to learn more about customization possibilities. Developers can customize the RSocketStrategies component by creating beans that implement the RSocketStrategiesCustomizer interface. Note that your @Order important because it determines the order of codecs. Spring Boot provides automatic configuration of the RSocket server. The required dependencies are provided by spring-boot-starter-rsocket. Spring Boot allows you to expose RSocket through WebSocket from a WebFlux server or to stand a separate RSocket server. This depends on the type of application and its configuration. For the WebFlux application (that is. Of type WebApplicationType.REACTIVE), the RSocket server will connect to the Web server only if the following properties match: spring.rsocket.server.mapping-path-/rsocket - a spring.rsocket.server.transport-websocket - websocket mapping path is defined as a #spring.rsocket.server.port transport - no Plug RSocket port is defined on a web server is only supported with Reactor Netty, as RSocket is constructed with that library. Alternatively, a TCP or websocket RSocket server starts as a stand-alone, embedded server. In addition to dependency requirements, the only configuration required is to define a port for that server: spring.rsocket.server.port-9898, the only configuration required spring.rsocket.server.transport-tcp, you can configure other properties that Spring Boot will automatically configure the spring messaging infrastructure for RSocket. This means that Spring Boot will create an RSocketMessageHandler bean that will handle RSocket requests to the application. Once the RSocket channel is established between server and client, either party can send or receive requests to the other. As a server, it can be inserted with an RSocketRequester instance into any handler method of an @Controller RSocket. As a client, you must first configure and establish an RSocket connection. Spring Spring auto configures an RSocketRequester.Builder for these cases with the expected codecs. The RSocketRequester.Builder instance is a prototype bean, which means that each injection point will provide you with a new instance. This is done on purpose because this generator is stateless and you should not create requesters with different configurations with the same instance. The following code shows a typical example: @Service myService - final private Mono<RSocketRequester> rsocketRequester; Public MyService(RSocketRequester.Builder rsocketRequesterBuilder) to this.rsocketRequester á rsocketRequesterBuilder .connectTcp(example.org, 9898).cache(); a public Mono<User> someRSocketCall(String name) ? return this.rsocketRequester.flatMap(req -> req.route(user). If Spring Security is in the class path, web applications are protected by default. Spring Boot relies on Spring Security's content negotiation strategy to determine whether to use httpBasic or formLogin. To add method-level security to a web application, you can also add @EnableGlobalMethodSecurity with the desired settings. Additional information can be found in the Spring Safety Reference Guide. The default UserDetailsService has a single user. The user name is user and the password is random and printed at the INFO level when the application starts, as shown in the following example: Using the generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35 If you adjust the logging settings, make sure that the org.springworkframe.boot.autoconfigure.security category is set in info-level messages. Otherwise, the default password is not printed. You can change the user name and password by providing a spring.security.user.name spring.security.user.password. The basic features that are obtained by default in a web application are: a UserDetailsService bean (or ReactiveUserDetailsService in case of a WebFlux application) with an in-memory store and a single user with a generated password (see SecurityProperties.User for user properties). Form-based login or HTTP Basic security (depending on the Accept header of the request) for the entire application (including actuator endpoints if the actuator is in the class path). A DefaultAuthenticationEventPublisher to publish authentication events. You can provide a different AuthenticationEventPublisher by adding a bean for it. The default security settings are implemented in SecurityAutoConfiguration and UserDetailsServiceAutoConfiguration. SecurityAutoConfiguration imports SpringBootWebSecurityConfiguration for web security and UserDetailsServiceAutoConfiguration configures authentication, which is also relevant in non-web applications. Completely disable the default web application security settings or combine multiple Spring Security components, such as OAuth 2 Client and Resource Server, add a Bean of type WebSecurityConfigurerAdapter (doing so does not disable UserDetailsService</User> </RSocketRequester> </RSocketRequester> or the safety of the actuator). To also disable UserDetailsService settings, you can add a bean of type UserDetailsService, AuthenticationProvider, or AuthenticationManager. Access rules can be overridden by adding a custom WebSecurityConfigurerAdapter. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. EndpointRequest can be used to create a RequestMatcher that is based on the management.endpoints.web.base-path property. PathRequest can be used to create a RequestMatcher for resources in commonly used locations. Like Spring MVC applications, you can secure WebFlux applications by adding the spring-boot-starter-security dependency. The default security settings are implemented in ReactiveSecurityAutoConfiguration and UserDetailsServiceAutoConfiguration. ReactiveSecurityAutoConfiguration imports WebFluxSecurityConfiguration for web security and UserDetailsServiceAutoConfiguration configures authentication, which is also relevant in non-web applications. To completely disable the default web application security settings, you can add a Bean of type WebFilterChainProxy (doing so does not disable UserDetailsService settings or actuator security). To also disable UserDetailsService settings, you can add a bean of type ReactiveUserDetailsService or ReactiveAuthenticationManager. Access rules and the use of various Spring Security components, such as OAuth 2 Client and Resource Server, can be configured by adding a custom SecurityWebFilterChain bean. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. EndpointRequest can be used to create a ServerWebExchangeMatcher that is based on the management.endpoints.web.base-path property. PathRequest can be used to create a ServerWebExchangeMatcher for resources in commonly used locations. For example, you can customize security settings by adding something like: @Bean securityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) - return http .authorizeExchange() .matchers(PathRequest.toStaticResources().atCommonLocations(), .lva() .pathMatchers(/foo, /bar) .authenticated(). OAuth2 is a widely used authorization framework that is compatible with Spring. If you have spring-security-oauth2-client in your class path, you can take advantage of some automatic configuration to configure an OAuth2/Open ID Connect client. This setting makes use of the OAuth2ClientProperties properties. The same properties apply to both servlet and reactive applications. You can register multiple OAuth2 clients and providers under the prefix spring.security.oauth2.client, shown in the following example: spring.security.oauth2.client.registration.my-client-1.client-id-abcd spring.security.oauth2.client.registration.my-client-1.client-secret-password spring.security.oauth2.client.registration.my-client-1.client-secret-password for user scope spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider spring.security.oauth2.client.registration.my-client-1.scope=user spring.security.oauth2.client.registration.my-client-1.redirect-uri= spring.security.oauth2.client.registration.my-client-1.client-authentication-method=basic spring.security.oauth2.client.registration.my-client-1.authorization-grant-type=authorization_code spring.security.oauth2.client.registration.my-client-2.client-id=abcd spring.security.oauth2.client.registration.my-client-2.client-secret=password spring.security.oauth2.client.registration.my-client-2.client-name=Client for email scope spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider spring.security.oauth2.client.registration.my-client-2.scope=email spring.security.oauth2.client.registration.my-client-2.redirect-uri= spring.security.oauth2.client.registration.my-client-2.client-authentication-method=basic spring.security.oauth2.client.registration.my-client-2.authorization-grant-type=authorization_code spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri= spring.security.oauth2.client.provider.my-oauth-provider.token-uri= spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri= spring.security.oauth2.client.provider.my-oauth-provider.user-info-authentication-method=header spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri= spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name For OpenID Connect providers that support OpenID Connect discovery, the configuration can be further simplified. The provider must be configured with an issuer-uri that is the URI that it claims as its issuer ID. For example, if the issuer-uri provided is , an OpenID provider configuration request will be made on . The result is expected to be an OpenID provider configuration response. The following example shows how an OpenID Connect provider can be configured with the issuer-uri: spring.security.oauth2.client.provider.oidc-provider.issuer-uri- By default, Spring Security OAuth2LoginAuthenticationFilter only processes URLs that match /login/oauth2/code/. If you want to customize the redirect-uri to use a different pattern, you must provide configuration to process that custom pattern. For example, for servlet applications, you can add your own WebSecurityConfigurerAdapter similar to the following: the OAuth2LoginSecurityConfig public class extends WebSecurityConfigurerAdapter @Override protected void configure(HttpSecurity http) throws . http .authorizeRequests() .anyRequest().authenticated() .anyRequest().authenticated() .oauth2Login() .redirectionEndpoint() .baseUri(/custom-callback); For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (google, github, facebook, and okta, respectively). If you do not need to customize these providers, you can set the provider attribute in which you need to infer default values. In addition, if the client registry key matches a default supported provider, Spring Boot also infers it. In other words, the two configurations in the following example use the Google provider: spring.security.oauth2.client.registration.my-client.client-id-abcd spring.security.oauth2.client.registration.my-client-secret-password spring.security.oauth2.client.registration.my-client.provider-google spring.security.oauth2.client.registration.google.client-id-abcd spring.security.oauth2.client.registration.google.client-secret-password If you have spring-security-oauth-resource-server in your path, Spring Boot can configure an OAuth2 resource server. For JWT configuration, you need to specify a JWK set URI or an OIDC issuer URI, As shown in the following examples: spring.security.oauth2.resourceserver.jwt.jwk-set-uri- spring.security.oauth2.resourceserver.jwt.issuer-uri- If the authorization server does not support a JWK Set URI, You can configure the resource server with the public key used to verify the JWT signature. This can be done by using the spring.security.oauth2.resourceserver.jwt.public-key-location property, where the value must point to a file that contains the public key in the PEM-encoded x509 format. The same properties are applicable for both servlet and reactive applications. Alternatively, you can define your own JwtDecoder bean for servlet applications or an ReactiveJwtDecoder for reactive applications. In cases where opaque tokens are used instead of JWT, You can configure the following properties to validate tokens through introspection: spring.security.oauth2.resourceserver.opaquetoken.introspection-uri- spring.security.oauth2.resourceserver.opaquetoken.client-id-my-client-id spring.security.oauth2.resourceserver.opaque.client-secret-my-client-secret Again, the same properties are applicable for both applications. Alternatively, you can define your own OpaqueTokenIntrospector bean for servlet applications or an ReactiveOpaqueTokenIntrospector for reactive applications. Spring Security does not currently provide support for deploying an OAuth 2.0 authorization server. However, this functionality is available in the Spring Security OAuth project, which will eventually be replaced by Spring Security completely. Then, you can use the spring-security-oauth2-autoconfigure module to easily configure an OAuth 2.0 authorization server; see your documentation for instructions. If you Spring-security-saml2-service-provider in your class path, you can take advantage of some automatic configuration to configure a SAML 2.0 trusted user. This setting makes use of saml2RelyingPartyProperties properties. A trusted user record represents a peered configuration between an identity provider, IDP, and a service provider, SP. You can register multiple relying parties under the spring.security.saml2.relyingparty prefix, as shown in the following example: spring.security.saml2.relyingparty.registration.my-relying-party1.signing.credentials[0].private-key-location=path-to-private-key spring.security.saml2.relyingparty.registration.my-relying-party1.signing.credentials[0].certificate-location=path-to-certificate spring.security.saml2.relyingparty.registration.my-relying-party1.identityprovider.verification.credentials[0].certificate-location=path-to-verification-cert spring.security.saml2.relyingparty.registration.my-relying-party1.identityprovider.entity-id=remote-idp-entity-id1 spring.security.saml2.relyingparty.registration.my-relying-party1.identityprovider.sso-url= spring.security.saml2.relyingparty.registration.my-relying-party2.signing.credentials[0].private-key-location=path-to-private-key spring.security.saml2.relyingparty.registration.my-relying-party2.signing.credentials[0].certificate-location=path-to-certificate spring.security.saml2.relyingparty.registration.my-relying-party2.identityprovider.verification.credentials[0].certificate-location=path-to-other-verification-cert spring.security.saml2.relyingparty.registration.my-relying-party2.identityprovider.entity-id=remote-idp-entity-id2 spring.security.saml2.relyingparty.registration.my-relying-party2.identityprovider.sso-url= For security purposes, all actuators other than /health and /info are disabled by default. The management.endpoints.web.exposure.include property can be used to enable actuators. If Spring Security is in the class path and no other WebSecurityConfigurerAdapter is present, all actuators other than /health and /info are protected by Spring Boot automatic configuration. If you define a custom WebSecurityConfigurerAdapter, spring boot auto-configuration will shut down and you have full control of the actuator access rules. Before setting management.endpoints.web.exposure.include, make sure that the exposed actuators do not contain sensitive information and/or are protected by placing it behind a firewall or by something like Spring Security. Because Spring Boot is based on Spring Security defaults, CSRF protection is turned on by default. This means that actuator endpoints that require a POST (shutdown endpoints and loggers), PUT or DELETE will get a prohibited 403 error when setting up security is in use. We recommend that you completely disable CSRF protection only if you are creating a service that uses non-browser clients. The Spring Framework provides broad support to the with SQL databases, from direct JDBC access using JdbcTemplate to object-relational mapping technologies such as Hibernate. Spring Data provides an additional level of functionality: create repository implementations directly from interfaces and use conventions to generate queries from method names. The Java javax.sql.DataSource interface provides a standard method for working with database connections. Traditionally, a 'DataSource' uses a URL along with some credentials to establish a database connection. See the How to do this section for more advanced examples, typically to take full control over DataSource settings. It is often convenient to develop applications by using an in-memory-embedded database. Obviously, in-memory databases do not provide persistent storage. You must populate the database when the application starts and be prepared to discard the data when the application finishes. The How to Do This section includes a section on how to initialize a database. Spring Boot can automatically configure built-in H2, HSQL, and Derby databases. You do not need to provide a connection URL. You only need to include a build dependency on the embedded database that you want to use. If you use this feature in your tests, you might notice that the entire test suite reuses the same database, regardless of the number of application contexts you use. If you want to ensure that each context has a separate embedded database, you must set spring.datasource.generate-unique-name to true. For example, typical POM dependencies would be as follows: <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> You need a dependency on spring-jdbc for an embedded database to be automatically configured. In this example, it is extracted transitively through spring-boot-starter-data-jpa. If, for any reason, you configure the connection URL for an embedded database, be careful to ensure that automatic database shutdown is disabled. If you use H2, you must use DB_CLOSE_ON_EXIT-FALSE to do so. If you are using HSQLDB, you must ensure that shutdown-true is not used. Disabling automatic database shutdown allows spring Boot control when the database is closed, ensuring that it occurs once database access is no longer required. Production database connections can also be configured using a pool data source. Spring Boot uses the following algorithm to choose a specific implementation: We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it. Otherwise, if the Tomcat Pool DataSource is available, we use it. If neither HikariCP nor the Tomcat pool data source is available and if Commons DBCP2 is available, we use it. If you use spring-boot-starter-jdbc or initiators, you automatically get a dependency on HikariCP. You can omit that algorithm completely and specify the connection pool to use by setting the spring.datasource.type property. This is especially important if you run your application in a Tomcat container, because tomcat-jdbc is provided by default. Additional connection pools can always be configured manually. If you define your own DataSource bean, automatic configuration does not occur. DataSource settings are controlled by external configuration properties in spring.datasource.*. For example, you can declare the following section in application.properties: spring.datasource.url-jdbc:mysql://localhost/test spring.datasource.username_dbuser spring.datasource.password-dbpass spring.datasource.driver-class-name-com.mysql.jdbc.Driver You must specify at least the URL by setting the spring.datasource.url property. Otherwise, Spring Boot attempts to automatically configure an embedded database. It is often not necessary to specify the driver-class-name, as Spring Boot can infer it for most databases in the url. To create a grouping dataSource, we need to be able to verify that a valid Driver class is available, so we verify that there is before we do anything. In other words, if you set spring.datasource.driver-class-name.com.mysql.jdbc.Driver, that class has to be loadable. See DataSourceProperties for more of the supported options. These are the standard options that work independently of the actual deployment. You can also adjust implementation-specific settings using their respective prefix (spring.datasource.hikari.*, spring.datasource.tomcat.* and spring.datasource.dbcp2.*). See the documentation for deploying the connection pool you are using for more details. For example, if you use the Tomcat connection pool, you can customize many additional configuration settings, as shown in the following example: . spring.datasource.tomcat.max-wait-10000 - Maximum number of active connections that can be allocated from this group at the same time. spring.datasource.tomcat.max-active-50 - Validate the connection before borrowing it from the group. spring.datasource.tomcat.test-on-borrow-true If you deploy the Spring Boot application to an application server, you may want to configure and manage the dataSource using the built-in features of the application server 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 of application.properties shows how you can access a source JBoss AS-defined data: spring.datasource.jndi-name-java:jboss/datasources/customers Spring's JdbcTemplate and NamedParameterJdbcTemplate classes are automatically configured, and you can @Autowire them, them, in their own beans, as shown in the following example: 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 name MyBean(JdbcTemplate jdbcTemplate) - this.jdbcTemplate - jdbcTemplate; You can customize some template properties by using the spring.jdbc.template.* properties, as shown in the following example: spring.jdbc.template.max-rows-500 The NamedParameterJdbcTemplate reuses the same jdbcTemplate instance in the background. If more than one JdbcTemplate is defined and no primary candidate exists, NamedParameterJdbcTemplate is not configured automatically. The Java Persistence API is a standard technology that allows you to map objects to relational databases. The POM spring-boot-starter-data-jpa provides a quick way to get started. It provides the following key dependencies: Hibernate: One of the most popular JPA implementations. Spring Data JPA: Helps you deploy JPA-based repositories. Spring ORM: Basic SPRING Frame ORM support. Traditionally, JPA Entity classes are specified in a persistence.xml file. With Spring Boot, this file is not required and Entity Scanning is used instead. By default, all packets are searched under the primary configuration class (the one that is annotated with @EnableAutoConfiguration or @SpringBootApplication). All classes annotated with a @Entity, @Embeddable, or @MappedSuperclass. A typical feature class is similar to the following example: package com.example.myapp.domain; import java.io.Serializable; import javax.persistence.*; @Entity City public class implements Serializable , @Id @GeneratedValue private Long id; @Column(nullable - false) private String name; @Column(nullable - false) private String state; // ... additional members often include @OneToMany City() protected assignments? no-args constructor required by JPA spec // this is protected since it should not be used directly - public City(String name, String state) - this.name ? name; this.state ? state; a public String getName() ? return this.name; ? public String getState() ? return this.state; ? Etc. Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from 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 the method with the Spring Data Query annotation. Spring Data repositories are typically extended from the Repository or CrudRepository interfaces. If you use automatic configuration, repositories are searched from the package that contains main configuration class (the one that is annotated @EnableAutoConfiguration or @SpringBootApplication) down. The following example shows a typical Spring Data Data repository definition: com.example.myapp.domain package; import org.springframework.data.domain.*; import org.springframework.data.repository.*; Public Interface CityRepository extends repository<City, long-> - Page<City> findAll(Pageable pageable pageable); City findByNameAndStateAllIgnoringCase(String name, String state); • Spring Data JPA repositories support three different boot modes: default, deferred, and lazy. To enable deferred or deferred booting, set the spring.data.jpa.repositories.bootstrap-mode property to deferred or deferred, respectively. When using deferred or deferred boot, automatically configured EntityManagerFactoryBuilder will use the AsyncTaskExecutor context, if any, as the boot executor. If more than one exists, the name applicationTaskExecutor will be used. By default, JPA databases are created automatically only if you use an embedded database (H2, HSQL, or Derby). You can explicitly configure JPA settings by using the spring.jpa.* properties. For example, to create and drop tables, you can add the following line to application.properties: spring.jpa.hibernate.ddl-auto-create-drop Hibernate's own internal property name for this (if best remembered) is hibernate.hbm2ddl.auto. You can set it, along with other native Hibernate properties, using spring.jpa.properties.* (the prefix is deleted before you add them to the entity manager). The following line shows an example of configuring JPA properties for Hibernate: spring.jpa.properties.hibernate.globally_quoted_identifiers-true The line in the previous example passes a value of true for the hibernate.globally_quoted_identifiers property to the Hibernate entity manager. By default, DDL execution (or validation) is deferred until ApplicationContext has started. There is also a spring.jpa.generate-ddl flag, but it is not used if the automatic Hibernate setting is active, because the ddl-auto settings are more accurate. If you are running a web application, Spring Boot registers OpenEntityManagerInViewInterceptor by default to apply the Open EntityManager in View pattern, to allow lazy loading into web views. If you do not want this behavior, you must set spring.jpa.open-in-view to false in application.properties. Spring Data includes repository support for JDBC and will automatically generate SQL for CrudRepository methods. For more advanced queries, an annotation is provided @Query. Spring Boot will automatically configure Spring Data JDBC repositories when the required dependencies are in the class path. They can be added to the project with a single dependency on spring-boot-starter-data-jdbc. If necessary, you can take control of Spring Data's JDBC configuration by adding the @EnableJdbcRepositories or a subclass to the app. The H2 database provides a browser-based console that Spring Boot can configure automatically. The console is automatically configured when the following conditions are met: Yes</City> </City,> </City,> Using spring boot development tools, but you would still like to make use of the H2 console, you can configure the spring.h2.console.enabled property with a value of true. The H2 console is only intended for use during development, so be careful to ensure that spring.h2.console.enabled is not set to true in production. By default, the console is available in /h2-console. You can customize the console path by using the spring.h2.console.path property. jOOQ Object Oriented Querying (jOOQ) is a popular Data Geekery product that generates Java code from the database and allows you to create type-safe SQL queries through its fluent API. Both commercial and open source editions can be used with Spring Boot. To use secure queries for jOOQ types, you must generate Java classes from the database schema. You can follow the instructions in the jOOQ user manual. If you use the jooq-codegen-maven plugin and also use the spring-boot-starter-parent parent POM, you can safely omit the <version>plugin tag. You can also use version variables defined by Spring Boot (such as h2.version) to declare dependency on the add-in database. The following list shows 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.org.h2.Driver</version> </dependency> </dependencies> <configuration> <jdbc> <driver></driver> <url>jdbc:h2:-/yourdatabase</url> </jdbc> <generator>... </generator> </configuration> </plugin> The fluent API offered by jOOQ is started through the org.jooq.DSLContext interface. Spring Boot automatically configures a DSLContext as a Spring Bean and connects it to the DataSource application. To use the DSLContext, you @Autowire, as shown in the following example: @Component JooqExample public class implements CommandLineRunner - private final DSLContext create; @Autowired public JooqExample(DSLContext dslContext) - this.create - dslContext; ? The jOOQ manual tends to use a variable named create to contain DSLContext. You can then use the DSLContext to construct the queries, as shown in the following example: 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); • Unless the spring.jooq.sql-dialect property is set, Spring Boot determines the SQL dialect to use for the source Data. If Spring Boot could not detect the dialect, use DEFAULT. Spring Boot can only automatically configure dialects that support the open source version of jOOQ. More advanced customizations can be achieved by defining your own @Bean definitions, which are used when you create the jOOQ configuration. You can define beans for the following jOOQ types: ConnectionProvider ExecutorProvider TransactionProvider RecordMapperProvider</GregorianCalendar> </version> </version> Settings RecordListenerProvider ExecuteListenerProvider VisitListenerProvider TransactionListenerProvider you can also create your own @Bean org.jooq.Configuration if you want to take full control of the jOOQ configuration. The Reactive Relational Database Connectivity (R2DBC) project brings reactive programming APIs to relational databases. The R2DBC io.r2dbc.spi.Connection connection provides a standard method for working with non-blocking database connections. Connections are provided through a ConnectionFactory, similar to a DataSource with jdbc. ConnectionFactory settings are controlled by external configuration properties in spring.r2dbc.*. For example, you can declare the following section in application.properties: spring.r2dbc.url-r2dbc:postgresql://localhost/test spring.r2dbc.username-dbuser spring.r2dbc.password-dbpass You do not need to specify a driver class name, because Spring Boot gets the R2DBC connection factory discovery driver. At least the url must be provided. The information specified in the URL takes precedence over individual properties, that is, name, user name, password, and grouping options. The How to Do This section includes a section on how to initialize a database. To customize connections created by ConnectionFactory, that is, set specific parameters that you do not want (or cannot) configure in the central database configuration, you can use a ConnectionFactoryOptionsBuilderCustomizer @Bean. The following example shows how to manually replace the database port, while the rest of the options are taken from the application configuration: @Bean connectionFactoryOptionsBuilderCustomizer connectionFactoryPortCustomizer() á return (builder) -> builder.option(PORT, 5432); The following examples show how to set some PostgreSQL connection options: @Bean Public ConnectionFactoryOptionsBuilderCustomizer postgresCustomizer() - Map Options<String, string-> - New HashMap <> (); options.put(lock_timeout, 30s); options.put(statement_timeout, 60s); return (builder) -> builder.option(OPTIONS, options); When a ConnectionFactory bean is available, the normal JDBC DataSource auto-configuration is disabled. If you want to preserve automatic JDBC DataSource settings and are comfortable with the risk of using the locking JDBC API in a reactive application, add @Import(DataSourceAutoConfiguration.class) to an application @Configuration class to re-enable it. Similar to JDBC support, Spring Boot can automatically configure an embedded database for reactive use. You do not need to provide a connection URL. You just need to include a build dependency on the database you want to use, as shown in the following example: <dependency> <groupId>io.r2dbc</groupId> <artifactId>r2dbc-h2</artifactId> <scope>runtime</scope> </dependency> If you use this feature in tests, you may notice that the entire database reuses the entire test suite regardless of the number of application contexts you use. If </String,> </String,> you want to ensure that each context has a separate embedded database, you must set spring.r2dbc.generate-unique-name to true. The Spring Data DatabaseClient class is automatically configured and can @Autowire directly into your own beans, as shown in the following example: import org.springframework.beans.factory.annotation.Autowired; import org.springframework.data.r2dbc.function.DatabaseClient; import org.springframework.stereotype.Component; @Component public class MyBean - private final DatabaseClient databaseClient; @Autowired public name MyBean(DatabaseClient databaseClient) - this.databaseClient ? databaseClient; ? // ... Spring Data R2DBC repositories are interfaces that you can define to access data. Queries are created automatically from 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 the method with the Spring Data Query annotation. Spring Data repositories are typically extended from the Repository or CrudRepository interfaces. If you use automatic configuration, repositories are searched from the package that contains the main configuration class (the one annotated with @EnableAutoConfiguration or @SpringBootApplication) down. The following example shows a typical spring Data repository interface definition: package com.example.myapp.domain; import org.springframework.data.domain.*; import org.springframework.data.repository.*; import reactor.core.publisher.Mono; Public Interface CityRepository extends Mono<City repository, long->City> findByNameAndStateAllIgnoringCase(String name, String state); Spring Data provides additional projects that help you access a variety of NoSQL technologies, including: Spring Boot provides automatic configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr Cassandra, Couchbase, and LDAP. You can make use of the other projects, but you must configure them yourself. Refer to the appropriate reference documentation spring.io/projects/spring-data. Redis is a cache store, message agent, and key-value store with rich functions. Spring Boot provides basic automatic configuration for lettuce and Jedi client libraries and abstractions above them provided by Spring Data Redis. There is a spring-boot-starter-data-redis Starter to collect dependencies in a convenient way. By default, use lettuce. That starter handles traditional, reactive applications. We also provide a spring-boot-starter-data-redis-reactive Starter for consistency with the other stores with reactive support. You can insert a configured vanilla RedisConnectionFactory, StringRedisTemplate, or RedisTemplate instance as I would any other Spring Bean. By default, the instance attempts to connect to a Redis server on localhost:6379. The following list shows an example of such a bean: @Component class MyBean - private StringRedisTemplate template; @Autowired public</City> </City,> </City,> template) - this.template - template; You can also register an arbitrary number of beans that implement LettuceClientConfigurationBuilderCustomizer for more advanced customizations. If you use Jedis, JedisClientConfigurationBuilderCustomizer is also available. If you add your own @Bean of any of the automatically configured types, it overrides the default value (except in the case of RedisTemplate, when the exclusion is based on the bean name, redisTemplate, not its type). By default, if commons-pool2 is in the class path, you get a pooled connection factory. MongoDB is an open source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several amenities to work with MongoDB, including spring-boot-starter-data-mongodb and spring-boot-starter-data-mongodb-reactive Starters. To access MongoDB databases, you can insert an automatic configuration organization org.springframework.data.mongodb.MongoDatabaseFactory. By default, the instance attempts to connect to a MongoDB server in mongodb://localhost/test. The following example shows how to connect to a MongoDB database: import org.springframework.data.mongodb.MongoDatabaseFactory; import com.mongodb.client.MongoDatabase; @Component public class MyBean á private final MongoDatabaseFactory mongo; @Autowired public name MyBean(MongoDatabaseFactory mongo) - this.mongo - mongo; ? // ... public void example() á MongoDatabase db a mongo.getMongoDatabase(); // ... You can set the spring.data.mongodb.uri property to change the URL and configure additional values, such as the replica set, as shown in the following example: spring.data.mongodb.uri-mongodb://user::12345,mongo2.example.com:23456/test Alternatively, you can specify connection details using discrete properties. For example, you can declare the following settings in your application.properties: spring.data.mongodb.host-mongoserver.example.com spring.data.mongodb.port-27017 spring.data.mongodb.database-test spring.data.mongodb.username-user spring.data.mongodb.password-secret If you have defined your own MongoClient, it will be used to automatically configure an appropriate MongoFactory. The automatically configured MongoClient is created using MongoClientSettings. To adjust your settings, declare one or more MongoClientSettingsBuilderCustomizer beans. Each will be called in order with the MongoClientSettings.Builder that is used to compile mongoClientSettings. If spring.data.mongodb.port is not specified, the default value of 27017 is used. You can remove this line from the example shown above. If you do not use Spring Data MongoDB, you can insert a MongoClient bean instead of using MongoDatabaseFactory. If you want to take full control of MongoDB connection, you can also declare your own MongoDatabaseFactory or MongoClient bean. If you are using the reactive driver, Netty is required for SSL. Automatic settings set this factory automatically if Netty Netty available and the factory to be used has not already been customized. Spring Data MongoDB provides a MongoTemplate class that is very similar in its layout to Spring's JdbcTemplate. As with JdbcTemplate, Spring Boot automatically configures a bean to inject the template, as follows: import org.springframework.data.mongodb.core.MongoTemplate; import org.springframework.stereotype.Component; @Component public class MyBean á private final MongoTemplate mongoTemplate; public MyBean(MongoTemplate mongoTemplate) - this.mongoTemplate ? mongoTemplate; ? // ... See MongoOperations Javadoc for full details. Spring Data includes repository support for MongoDB. As with the JPA repositories discussed above, the basic principle is that queries are constructed automatically, based on method names. In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the previous JPA example, and assuming that City is now a MongoDB data class instead of a JPA @Entity, it works the same way, as shown in the following example: 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 pageable); City findByNameAndStateAllIgnoringCase(String name, String state); • You can customize document scanning locations using @EntityScan. For complete details of Spring Data MongoDB, including its rich object mapping technologies, see its reference documentation. Spring Boot offers automatic configuration for built-in Mongo. To use it in the Spring Boot application, add a dependency on de.flapdoodle.embed:de.flapdoodle.embed.mongo. The port on which Mongo listens can be configured by setting the spring.data.mongodb.port property. To use a randomly assigned free port, use a value of 0. The MongoClient created by MongoAutoConfiguration is automatically configured to use the randomly assigned port. If you do not configure a custom port, embedded support uses a random port (instead of 27017) by default. If you have SLF4J in the class path, the output produced by Mongo is 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 log routing. Download settings can be customized by declaring a DownloadConfigBuilderCustomizer bean. Neo4j is an open source NoSQL graphics database that uses a rich data model of nodes connected by class, which is more suitable for connected big data than traditional RDBMS approaches. Spring Boot offers several amenities to work with Neo4j, including the spring-boot-starter-data-neo4j Starter. To access a Neo4j server, you can insert an automatically configured org.neo4j.ogm.session.Session session. By default, the instance attempts</City> </City,> </City,> to a Neo4j server on localhost:7687 using the Bolt protocol. The following example shows how to inject a Neo4j session: @Component myBean public class - private final session session; @Autowired myBean(Session session) - this.session - session; ? // ... You can configure the uri and credentials to use by setting the spring.data.neo4j.* properties, As shown in the following example: spring.data.neo4j.uri-bolt://my-server:7687 spring.data.neo4j.username-neo4j spring.data.neo4j.password-secret You can take full control over session creation by adding a bean org.neo4j.ogm.config.Configuration or a bean org.neo4j.ogm.session.SessionFactory. If you add org.neo4j:neo4j-ogm-embedded-driver to application dependencies, Spring Boot automatically configures an embedded instance in the Neo4j process that does not retain any data when the application is closed. As the built-in Neo4j OGM driver does not provide the Neo4j kernel itself, you have to declare org.neo4j:neo4j as a dependency yourself. See the Neo4j OGM documentation for a list of supported versions. The embedded controller takes precedence over the other handlers when there are multiple handlers in the class path. You can explicitly disable embedded mode by setting spring.data.neo4j.embedded.enabled-false. Neo4j data tests automatically use an embedded Neo4j instance if the embedded driver and Neo4j kernel are in the class path as described above. You can enable persistence for embedded mode by providing a path to a database file in the configuration, for example, spring.data.neo4j.uri-file://var/tmp/graph.db. Neo4j-OGM can map some types, such as java.time.*, to string-based properties, or to one of the native types provided by Neo4j. For backward compatibility reasons, the default value for Neo4j-OGM is to use a string-based representation. To use native types, add a dependency on org.neo4j:neo4j-ogm-bolt-native-types or org.neo4j:neo4j-ogm-embedded-native-types, and configure the spring.data.neo4j.use-native-types property as shown in the following example: spring.data.neo4j.use-native-types-true By default, if you are running a web application, the session is thread-bound for all request processing (that is, you use the Open Session in View pattern). If you do not want this behavior, add the following line to the application.properties file: spring.data.neo4j.open-in-view-false Spring Data includes repository support for Neo4j. Spring Data Neo4j shares the common infrastructure with Spring Data JPA like many other spring Data modules. You could take the JPA example above and define City as Neo4j GMO @NodeEntity instead of JPA @Entity and abstraction of the works the same way, as shown in the following example: package com.example.myapp.domain; import java.util.Optional; import org.springframework.data.neo4j.repository.*; CityRepository public interface extends Neo4jRepository<City, long-> </City,> </City,> findOneByNameAndState(String name, String state); • Spring-boot-starter-data-neo4j Starter enables repository support as well as transaction management. You can customize locations to search for repositories and entities @EnableNeo4jRepositories and @EntityScan respectively in a @Configuration-bean. For complete details of Spring Data Neo4j, including its object mapping technologies, see the reference documentation. Apache Solr is a search engine. Spring Boot provides basic automatic configuration for the Solr 5 client library and abstractions provided by Spring Data Solr. There is a spring-boot-starter-data-solr Starter to collect dependencies in a convenient way. You can insert an automatically configured SolrClient instance as you would any other Spring bean. By default, the instance attempts to connect to a server in localhost:8983/solr. The following example shows how to inject a Solr bean: @Component public class MyBean a solrClient private solr; @Autowired myBean(SolrClient solr) - this.solr á solr; If you add your own @Bean of type SolrClient, it overrides the default value. Spring Data includes repository support for Apache Solr. As with the JPA repositories described above, the basic principle is that queries are constructed automatically based on method names. In fact, both Spring Data JPA and Spring Data Solr share the same common infrastructure. You could take the JPA example from before, and assuming City is now a @SolrDocument class rather than a jpA @Entity, it works the same way. Elasticsearch is an open source, distributed RESTful

search and analysis engine. Spring Boot provides basic automatic configuration for Elasticsearch. Spring Boot supports multiple clients: official low-level, high-level Java REST clients The ReactiveElasticsearchClient provided by Spring Data Elasticsearch Spring Boot provides a dedicated Starter, spring-boot-starter-data-elasticsearch. Elasticsearch sends two different REST clients that you can use to query a cluster: the low-level client and the high-level client. If you have the org.elasticsearch.client:elasticsearch-rest-client dependency on the class path, Spring Boot will automatically configure and register a RestClient bean that by default is targeted localhost:9200. You can further adjust how RestClient is configured, as shown in the following example: spring.elasticsearch.rest.uris- spring.elasticsearch.rest.read-timeout-10s spring.elasticsearch.rest.username-user spring.elasticsearch.rest.password-secret You can also register an arbitrary number of beans that implement RestClientClientCustomizer for more advanced customizations. To take full control over the define a RestClientBuilder bean. If you have the client dependency org.elasticsearch.client:elasticsearch-rest-high-level in the class path, Spring Boot will automatically configure a RestHighLevelClient, which takes advantage of any reusing your HTTP configuration. Spring Data Elasticsearch sends ReactiveElasticsearchClient to query Elasticsearch instances reactively. It is based on WebFlux WebClient, so the spring-boot-starter-elasticsearch and spring-boot-starter-webflux dependencies are useful for enabling this support. By default, Spring Boot will automatically configure and register an ReactiveElasticsearchClient bean that targets localhost:9200. You can further adjust how it is configured, as shown in the following example: spring.data.elasticsearch.client.reactive.endpoints-search.example.com:9200 spring.data.elasticsearch.client.reactive.use-ssl-true spring.data.elasticsearchsearch client.reactive.socket-timeout-10s spring.data.elasticsearch.client.reactive.username-user spring.data.elasticsearch.client.reactive.password-secret If the configuration properties are not sufficient and you want to fully control the client configuration, you can register a custom ClientConfiguration bean. To connect to Elasticsearch, a RestHighLevelClient bean must be automatically defined using Spring Boot or provided manually by the application (see the previous sections). With this configuration in place, an ElasticsearchRestTemplate can be injected like any other Spring bean, as shown in the following example: @Component public class MyBean - private final ElasticsearchRestTemplate template; Public MyBean(ElasticsearchRestTemplate template) - this.template - template; In the presence of spring-data-elasticsearch and the dependencies required to use a WebClient (typically spring-boot-starter-webflux), Spring Boot can also automatically configure an ReactiveElasticsearchClient and a ReactiveElasticsearchTemplate as beans. They are the reactive equivalent of other REST clients. Spring Data includes repository support for Elasticsearch. As with the JPA repositories described above, the basic principle is that queries are constructed automatically based on method names. In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure. You could take the JPA example from before, and assuming City is now an Elasticsearch @Document class instead of a JPA @Entity, it works the same way. Spring Boot supports classic and reactive Elasticsearch repositories, using ElasticsearchRestTemplate or ReactiveElasticsearchTemplate beans. Those beans are most likely automatically configured by Spring Boot because the required dependencies are present. If you want to use your own template to back up Elasticsearch repositories, you can add your own elasticsearchRestTemplate or ElasticsearchOperations @Bean, as long as elasticsearchTemplate is called. The same applies to ReactiveElasticsearchTemplate and with the name bean reactiveElasticsearchTemplate. You can choose to disable repository support with the following property: spring.data.elasticsearch.repositories.enabled-false Cassandra is a distributed database management system designed to handle large amounts of data on many commodity servers. Spring Boot offers self-configuration for Cassandra and abstractions above it provided by Spring Data Cassandra. There is a spring-boot-starter-data-cassandra Starter to collect dependencies in a convenient way. You can insert an instance of CassandraTemplate or Cassandra CqlSession automatically configured as you would any other Spring Bean. Spring.data.cassandra.* properties can be used to customize the connection. Usually, provides the key name and contact points, as well as the name of the local datacenter, as shown in the following example: spring.data.cassandra.keyspace-name-mykeyspace spring.data.cassandra.contact-points-cassandrahost1:9042,cassandrahost2:9042 spring.data.cassandra.local-datacenter-1 If the port is the same for all of your contact points, you can use a shortcut and specify only host names, as shown in the following example: spring.data.cassandra.keyspace-name-mykeyspace spring.data.cassandra.contact-points-cassandrahost1,cassandrahost2 spring.data.cassandra.local-datacenter-datacenter1 Those two examples are identical as the port default to 9042. If you need to configure the port, use spring.data.cassandra.port. The Cassandra driver has its own configuration infrastructure that loads an application.conf into the root of the class path. Spring Boot does not search for such a file and instead provides a number of configuration properties through the spring.data.cassandra.* namespace. For more advanced driver customizations, you can register an arbitrary number of beans that implement DriverConfigLoaderBuilderCustomizer. The CqlSession can be customized with a bean of type CqlSessionBuilderCustomizer. If you use CqlSessionBuilder to create multiple CqlSession beans, note that the generator is mutable, so be sure to insert a new copy for each session. The following code list shows how to inject a Cassandra bean: @Component public class MyBean - private final CassandraTemplate template; Public MyBean(CassandraTemplate template) - this.template - template; If you add your @Bean of type CassandraTemplate, it overrides the default value. Spring Data includes basic support for the Cassandra repository. Currently, this is more limited than the JPA repositories discussed above and you need to annotate search methods with @Query. Couchbase is an open source, distributed, and multi-model database oriented to NoSQL documents that is optimized for interactive applications. Spring Boot provides automatic configuration for Couchbase and abstractions above it provided by Spring Data Couchbase. There's and spring-boot-starter-data-couchbase-reactive Starters to collect dependencies in a convenient way. You can get a cluster by adding the Couchbase SDK and some configuration. Spring.couchbase.* properties can be used to customize the connection. Usually, generally, the connection string, user name, and password, as shown in the following example: spring.couchbase.connection-string-couchbase://192.168.1.123 spring.couchbase.username-user spring.couchbase.password-secret It is also possible to customize some of the ClusterEnvironment settings. For example, the following setting changes the timeout to be used 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. To take more control, you can use one or more ClusterEnvironmentBuilderCustomizer beans. Spring Data includes repository support for Couchbase. For complete Spring Data Couchbase details, see the reference documentation. You can insert an automatically configured CouchbaseTemplate instance as you would any other Spring Bean, as long as a CouchbaseClientFactory bean is available. This happens when a cluster is available, as described above, and a bucket name has been specified: spring.data.couchbase.bucket-name-my-bucket The following examples show how to inject a CouchbaseTemplate bean: @Component public class MyBean - couchbaseTemplate final private template; @Autowired my public template MyBean(BedbaseTemplate Template) - this.template - template; There are some beans that you can define in your own configuration to override those provided by automatic configuration: A CouchbaseMappingContext @Bean with a couchbaseMappingContext name. A CustomConversions @Bean with a name of couchbaseCustomConversions. A @Bean CouchbaseTemplate with the name couchbaseTemplate. To avoid hard-coding those names in your own configuration, you can reuse BeanNames provided by Spring Data Couchbase. For example, you can customize the converters to use, as follows: @Configuration(proxyBeanMethods - false) public class SomeConfiguration ? @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS) public CustomConversions myCustomConversions() - return new CustomConversions(...); Lightweight Directory Access Protocol (LDAP) is an industry-standard, provider-neutral application protocol for accessing and maintaining directory information services distributed over an IP network. Spring Boot provides automatic configuration for any supported LDAP server, as well as support for unboundID's in-memory LDAP server. LDAP abstractions are provided by Spring Data LDAP. There is a spring-boot-starter-data-ldap Starter to collect dependencies in a convenient way. To connect to an LDAP server, be sure to declare a dependency on Starter or spring-ldap-core, and then declare your server URLs in your application.properties, as shown in the following example: spring.ldap.urls-ldap://myserver:1235 spring.ldap.username-admin spring.ldap.password-secret If you need to customize the connection settings, you can use the connection settings , you can use the connection settings, you can use the connection settings, you can use the connection settings, you can use the connection settings, you can use the connection settings, you can use the connection settings, you can use the connection settings, you can use the settings and spring.ldap.base-environment properties. An LdapContextSource is automatically configured based on this setting. If a DirContextAuthenticationStrategy bean is available, it is associated with the automatically configured LdapContextSource. If you need to customize it, for example, to use a PooledContextSource, you can still insert the automatically configured LdapContextSource. Be sure to mark the custom ContextSource as @Primary for the automatically configured LdapTemplate to use. Spring Data includes repository support for LDAP. For complete Spring Data LDAP details, see the reference documentation. You can also insert an automatically configured LdapTemplate instance as you would any other Spring Bean, as shown in the following example: @Component public class MyBean - private final LdapTemplate template; @Autowired myBean(LdapTemplate template) public - this.template - template; For testing, Spring Boot supports automatic configuration of an in-memory LDAP server from UnboundID. To configure the server, add a dependency to com.unboundid:unboundid-ldapsdk and declare a spring.ldap.embedded.base-dn property, as follows: spring.ldap.embedded.base-dn-dc-spring,dc-io It is possible to define multiple base-dn values, however, because distinguished names typically contain commas, they must be defined by correct notation. In yaml files, you can use yaml list notation: spring.ldap.embedded.base-dn: - dc-spring,dc-io - dc-pivotal,dc-io In property files, you must include the index as part of the property name: spring.ldap.embedded.base-dn[0]-dc-spring,dc-io spring.ldap.embedded.base-dn[1]-dc-pivotal,dc-io By default, the server starts on a random port and triggers regular LDAP support. You do not need to specify a spring.ldap.urls property. If there is a schema.ldif file in the class path, it is used to initialize the server. If you want to load the initialization script from a different resource, you can also use the spring.ldap.embedded.ldif property. By default, a standard schema is used to validate LDIF files. You can turn off validation completely by setting the spring.ldap.embedded.validation.enabled property. If you have custom attributes, you can use spring.ldap.embedded.validation.schema to define custom attribute types or object classes. InfluxDB is an open source time series database optimized for fast, high-availability storage and recovery of time series data in fields such as operations monitoring, application metrics, Internet of Things sensor data, and real-time analytics. Spring Boot automatically configures an InfluxDB instance, as long as the influxdb-java client is in the and set the database URL, as shown in the following example: spring.influx.url- If the connection to InfluxDB requires a user and password, you can set the spring.influx.user and spring.influx.password properties accordingly. InfluxDB is based on OkHttp. If you need to adjust the http client InfluxDB uses behind the you can register an InfluxDbOkHttpClientBuilderProvider bean. Spring Framework provides support for transparently adding caching to an application. In essence, abstraction applies caching to methods, reducing the number of executions based on the information available in the cache. Caching logic is applied transparently, without any interference to the invoker. Spring Boot automatically configures the cache infrastructure as long as caching support is enabled through @EnableCaching. Simply put, to add caching to a service operation, add the relevant annotation to your method, as shown in the following example: import org.springframework.cache.annotation.Cacheable; import org.springframework.stereotype.Component; @Component mathService public class @Cacheable(piDecimals) public int computePiDecimal(int i) á // á . This example shows the use of caching in a potentially expensive operation. Before invoking computePiDecimal, abstraction looks for an entry in the piDecimals cache that matches the i argument. If an entry is found, the contents of the cache are immediately returned to the caller and the method is not invoked. Otherwise, the method is invoked and the cache is updated before the value is returned. You can also use standard JSR-107 (JCache) annotations (such @CacheResult) transparently. However, we strongly recommend that you do not mix and match the Spring Cache and JCache annotations. If you do not add a specific cache library, Spring Boot automatically configures a simple provider that uses concurrent maps in memory. When a cache is required (such as piDecimals in the previous example), this provider creates it for you. The simple vendor is not really recommended for production use, but it is ideal to get started and make sure you understand the features. When you have made a decision about the cache provider to use, be sure to read its documentation to find out how to configure the caches that your application uses. Almost all providers require you to explicitly configure each cache that you use in your application. Some offer a way to customize the default caches defined by the spring.cache.cache-names property. You can also transparently refresh or eject data from the cache. Cache abstraction does not provide a real store and is based on abstraction materialized by the org.springframework.cache.Cache and org.springframework.cache.CacheManager interfaces. If you have not defined a CacheManager bean or CacheResolver named cacheResolver (see CachingConfigurer), Spring Boot attempts to detect the following providers (in the indicated): It is also possible to force a particular cache provider by setting the spring.cache.type property. Use this property if you need to disable caching completely in certain environments (such as testing). Use spring-boot-starter-cache Starter to quickly add basic caching dependencies. The starter brings If you add dependencies manually, you must include spring-context-support before you can use JCache, EhCache 2.x, or Caffeine support. If the CacheManager is automatically configured by Spring Boot, you can further adjust its configuration before it is fully initialized by exposing a bean that implements the CacheManagerCustomizer interface. The following example sets a flag to say that null values must be passed to the underlying map: @Bean<ConcurrentMapCacheManager> cacheManagerCustomizer cacheManagerCustomizer() public , returns a new CacheManagerCustomizer<ConcurrentMapCacheManager>() , @Override public void customize(ConcurrentMapCacheManager cacheManager) , cacheManager.setAllowNullValues(false); In the previous example, an automatically configured ConcurrentMapCacheManager is expected. If that is not the case (whether you have provided your own configuration or a different cache provider was configured automatically), the customizer is not invoked at all. You can have as many customizers as you want, and you can also sort them @Order or Sorted. Generic caching is used if the context defines at least one org.springframework.cache.Cache bean. A CacheManager is created that wraps all beans of that type. JCache is booted through the presence of a javax.cache.spi.CachingProvider in the class path (that is, there is a JSR-107-compatible caching library in the class path) and JCacheCacheManager is provided by the spring-boot-starter-cache Starter. Several supported libraries are available, and Spring Boot provides dependency management for Ehcache 3, Hazelcast, and Infinispan. You can also add any other supported library. There may be more than one provider, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not apply a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate the configuration of a cache with deployment details, as shown in the following example: - It is only necessary if more than one provider is present spring.cache.jcache.provider.com.acme.MyCachingProvider spring.cache.jcache.config-classpath:acme.xml When a cache library provides native implementation and jsR-107 support, Spring Boot prefers JSR-107 support, so the same features are available if you switch to a different JSR-107 deployment. Spring Boot has general support for Hazelcast. If a single HazelcastInstance is available, it is also automatically reused for CacheManager, unless the spring.cache.jcache.config property is specified. There are two ways to customize the underlying javax.cache.cacheManager: caches can be created at startup by setting the 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 CacheManager reference for complete customization. If a standard</ConcurrentMapCacheManager> </ConcurrentMapCacheManager> </ConcurrentMapCacheManager> bean is defined, it is automatically adjusted in an org.springframework.cache.CacheManager implementation that expects abstraction. No additional customization is applied to you. EhCache 2.x is used if a file named ehcache.xml can be found at the root of the class path. If EhCache 2.x is found, the EhCacheCacheManager provided by the spring-boot-starter-cache Starter is used to boot the cache manager. You can also provide an alternate configuration file, as shown in the following example: spring.cache.ehcache.config-classpath:config/another-config.xml Infinispan has no default location of the configuration file, so it must be explicitly specified. Otherwise, the default boot is used. spring.cache.infinispan.config-infinispan.xml Caches can be created at startup by setting the spring.cache.cache-names property. If a custom ConfigurationBuilder bean is defined, it is used to customize caches. Infinispan support in Spring Boot is restricted to integrated mode and is quite basic. If you want more options, you should use the official Infinispan Spring Boot instead. See the Infinispan documentation for more details. If Spring Data Couchbase is available and Couchbase is configured, a CouchbaseCacheManager is configured automatically. You can create additional caches at startup by setting the spring.cache.cache-names property and the cache defaults can be configured using the spring.cache.couchbase.* properties. For example, the following configuration creates cache caches cache1 and cache2 with an input expiration of 10 minutes: spring.cache.cache-names-cache1,cache2 spring.cache.couchbase.expiration-10m If you need more control over the configuration, consider registering a CouchbaseCacheManagerCustomizer bean. The following example shows a customizer that configures a specific input expiration for cache1 and cache2: @Bean public CouchbaseCacheManagerBuilderCustomizer myCouchbaseCacheManagerBuilderCustomizer() - return (builder) -> .withCacheConfiguration(cache1, CouchbaseCacheConfiguration.defaultCacheConfig().entryExpiry(Duration.ofSeconds(10))) constructor) .withCacheConfiguration(cache2, CouchbaseCacheConfiguration.defaultCacheConfig().entryExpiry(Duration.ofMinutes(1))); If Redis is available and configured, a RedisCacheManager is configured automatically. Additional caches can be created at startup by setting the spring.cache.cache-names property and cache defaults can be configured using the spring.cache.redis.* properties. For example, the following configuration creates cache caches cache1 and cache2 with a lifespan of 10 minutes: spring.cache.cache-names-cache1,cache2 spring.cache.redis.time-to-live-600000 By default, a prefix of so that if two separate caches use the same key, Redis has no overlapping keys and cannot return invalid values. We strongly recommend that you keep this setting enabled if you create your own RedisCacheManager. You can take full control of the default settings by adding a @Bean on your own. This can be useful if you are looking to customize the default serialization strategy. If you need more control over the configuration, consider registering a RedisCacheManagerBuilderCustomizer bean. The following example shows a customizer that configures a specific time to live for cache1 and cache2: redisCacheManagerBuilderCustomizer public @Bean myRedisCacheManagerBuilderCustomizer() - return (builder) -> .withCacheConfiguration(cache1, RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofSeconds(10))) .withCacheConfiguration(cache2, RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofMinutes(1))); Caffeine is a Java 8 rewrite of guava's cache that replaces support for Guava. If caffeine is present, a CaffeineCacheManager (provided by the spring-boot-starter-cache Starter) is automatically configured. Caches can be created at startup by setting the spring.cache.cache-names property and can be customized by one of the following (in the order listed): a cache specification defined by spring.cache.caffeine.spec A com.github.benmanes.caffeine.cache.Caffeine Bean is defined A com.github.benmanes.caffeine.caffeine.cache.Caffeine bean the following setting creates cache1 and cache2 a maximum size of 500 and a lifespan of 10 minutes spring.cache.cache-names-cache1,cache2 spring.cache.caffeine.spec-maximumSize-500,expireAfterAccess-600s If you define a bean com.github.benmanes.caffeine.cache.CacheLoader, it is automatically associated with the bean com.github.benmanes.caffeine.cache.CacheLoader, is automatically associated with the bean com.github.benmanes.caffeine.cache.CacheLoader, is automatically associated with the bean com.github.benmanes.caffeine.cache.CacheLoader, it is automatically associated with the bean com.github.benmanes.caffeine.cache.CacheLoader, it is automatically associated with the bean com.github.benmanes.caffeine.cache.CacheLoader, it is automatically associated with the bean. Because CacheLoader is to be associated with all caches managed by the cache manager, it must be set to CacheLoader<Object, object->. Automatic configuration ignores any other generic types. If none of the other providers can be found, a simple implementation is configured using ConcurrentHashMap as the cache store. This is the default if no caching library is present in the application. By default, caches are created as needed, but you can restrict the list of available caches by setting the cache-names property. For example, if you only want cache1 and cache2, set the cache-names property as follows: spring.cache.cache-names-cache1,cache2 If you do and your application uses a cache that is not listed, a run-time error occurs when memory is needed but not at the beginning. This is similar to how actual cache providers behave if you use an undeclared cache. When @EnableCaching is present in your configuration, an appropriate cache configuration is also expected. If you need to disable caching completely in certain environments, force the cache type to none to use a non-op implementation, as shown in the following example: Spring Framework provides extensive support for integration 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 </Object,> </Object,> Queue protocol. Spring Boot also provides automatic configuration options for RabbitTemplate and RabbitMQ. Spring WebSocket natively includes support for STOMP messaging, and Spring Boot supports that through starters and a small amount of automatic configuration. Spring Boot also has support for Apache Kafka. The javax.jms.ConnectionFactory interface provides a standard method for creating a javax.jms.Connection to interact with a JMS broker. Although Spring needs a ConnectionFactory to work with JMS, you generally don't need to use it directly yourself and can instead rely on top-level messaging abstractions. (See the relevant section of the Spring Framework reference documentation for more information.) Spring Boot also automatically configures the infrastructure needed to send and receive messages. When ActiveMQ is available in the class path, Spring Boot can also configure ConnectionFactory. If the broker is present, an embedded broker starts and configures automatically (as long as no broker URLs are specified through the configuration). If you use spring-boot-starter-activemq, the dependencies required to connect or embed an activeMQ instance are provided, as is the spring infrastructure to integrate with JMS. ActiveMQ settings are controlled by external configuration properties in spring.activemq.*. For example, you can declare the following section in application.properties: spring.activemq.broker-url-tcp://192.168.1.210:9876 spring.activemq.user-admin spring.activemq.password-secret By default, a CachingConnectionFactory wraps the native ConnectionFactory with reasonable values that you can control by using external configuration properties in spring.jms.*: spring.jms.cache.session-cache-size-5 If you prefer to use native pooling, You can do this by adding a dependency to org.messaginghub:pooled-jms and configuring JmsPoolConnectionFactory accordingly, as shown in the following example: spring.activemq.pool.enabled-true spring.activemq.pool.max-connections-50 See ActiveMQProperties for more of the supported options. You can also register an arbitrary number of beans that implement ActiveMQConnectionFactoryCustomizer for more advanced customizations. By default, ActiveMQ creates a destination if it does not already exist so that the targets are resolved with the provided names. Spring Boot can automatically configure a ConnectionFactory when it detects that Artemis is available in the class path. If the broker is present, an embedded broker is automatically started and configured (unless the mode property is explicitly set). Supported modes are embedded (to make it explicit that a embedded and that an error must occur if the broker is not available in the class path) and native (to connect to a broker using the netty transport protocol). When the latter is configured, Spring Boot configures a ConnectionFactory that connects to a broker running on the local computer with the default settings. If you use spring-boot-starter-artemis, spring-boot-starter-artemis, the dependencies required 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 the application allows you to use embedded mode. Artemis settings are controlled by external configuration properties in spring.artemis.*. For example, you can declare the following section in application.properties: spring.artemis.mode-native spring.artemis.host-192.168.1.210 spring.artemis.port-9876 spring.artemis.user-admin spring.artemis.password-secret When embedding the broker, you can choose whether to enable persistence and list the destinations that should be available. These can be specified as a comma-separated list to create with the default options, or you can define bean(s) of type org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration or org.apache.activemq.artemis.jms.server.config.TopicConfiguration, for advanced queue and topic configurations, respectively. By default, a CachingConnectionFactory wraps the native ConnectionFactory with reasonable values that you can control by external configuration properties in spring.jms.*: spring.jms.cache.session-cache-size-5 If you prefer to use native grouping, You can do this by adding a dependency to org.messaginghub:pooled-jms and configuring JmsPoolConnectionFactory accordingly, as shown in the following example: spring.artemis.pool.enabled-true spring.artemis.pool.max-connections-50 See ArtemisProperties for more supported options. There is no JNDI lookup involved, and targets are resolved with their names, using the name attribute in the Artemis configuration or the names provided through the configuration. If you are running the application on an application server, Spring Boot attempts to locate a JMS ConnectionFactory using JNDI. By default, the java:/JmsXA and java:/XAConnectionFactory location are checked. You can use the spring.jms.jndi-name property if you need to specify an alternate location, as shown in the following example: spring.jms.jndi-name-java:/MyConnectionFactory Spring's JmsTemplate is automatically configured, and you can automatically connect it directly to your own beans, as shown in the following example: import org.springframework.beans.factory.annotation.Autowired; import org.springframework.jms.core.JmsTemplate; import org.springframework.stereotype.Component; @Component public class MyBean a final private JmsTemplate jmsTemplate; @Autowired public name MyBean(JmsTemplate jmsTemplate) - this.jmsTemplate ? jmsTemplate; JmsMessagingTemplate can be injected in a similar way. If a DestinationResolver or MessageConverter bean is defined, it is automatically associated with the automatically configured JmsTemplate. When JMS infrastructure is present, any bean can be with @JmsListener to create a listener endpoint. If no JmsListenerContainerFactory is defined, a default JmsListenerContainerFactory is automatically configured. If a DestinationResolver or a beans are defined, it is automatically associated with the default factory. By default, the default factory is transactional. If you run on an infrastructure where a JtaTransactionManager is present, it is associated with the listener container by default. If not, the sessionTransacted flag is enabled. In the latter scenario, you can associate the local datastore transaction with the processing of an incoming message by adding @Transactional in the listener method (or a delegate from it). This ensures that the incoming message is recognized after the local transaction is complete. This also includes sending response messages that have been made in the same JMS session. The following component creates a listener endpoint on the someQueue target: @Component public class MyBean á @JmsListener(destination ? someQueue) public void processMessage(String content) to // ... See the @EnableJms Javadoc for more details. If you need to create more instances of JmsListenerContainerFactory or if you want to override the default value, Spring Boot provides a DefaultJmsListenerContainerFactoryConfigurer that you can use to initialize a DefaultJmsListenerContainerFactory with the same settings as the one that is automatically configured. For example, the following example exposes another generator that uses a specific MessageConverter: @Configuration(proxyBeanMethods) static class JmsConfiguration (new DefaultJmsListenerContainerFactory(@Bean public DefaultJmsListenerContainerFactory myFactory(DefaultJmsListenerContainerFactoryConfigurer configurer) ? DefaultJmsListenerContainerFactory factory ? new DefaultJmsListenerContainerFactory(); configurer.configure(factory, connectionFactory()); factory.setConverterMessage()) You can then use the factory in any @JmsListener-annotized method as follows: @Component my public class MyBean ? @JmsListener(destination) someQueue, containerFactory myFactory) public void processMessage(String content) ? Advanced Message Queuing Protocol (AMQP) is a platform-neutral cable-level protocol for message-oriented middleware. The Spring AMQP project applies Spring basics to the development of AMQP-based messaging solutions. Spring Boot offers several amenities to work with AMQP through RabbitMQ, including the spring-boot-starter-amqp Starter. RabbitMQ is a lightweight, reliable, scalable and portable message agent based on the AMQP protocol. Spring uses RabbitMQ to communicate through the AMQP protocol. RabbitMQ settings are controlled by external configuration properties in spring.rabbitmq.*. For example, you can declare the following section in application.properties: spring.rabbitmq.host-localhost spring.rabbitmq.port-5672 spring.rabbitmq.username-admin spring.rabbitmq.password-secret Alternatively, you can configure the same using the addresses attribute: spring.rabbitmq.addresses-amqp://admin: When specifying addresses that way, the host and port port are ignored. If the address uses the amqps protocol, SSL support is enabled automatically. If a ConnectionNameStrategy bean exists in the context, it will be automatically used to name connections created by the automatically configured ConnectionFactory. See RabbitProperties for more of the supported options. See Understanding AMQP, the protocol used by RabbitMQ for more details. Spring AmqpTemplate and AmqpAdmin are automatically configured, and you can automatically connect them directly to your own beans, as shown in the following example: 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 a final private AmqpAdmin amqpAdmin; final private AmqpTemplate amqpTemplate; @Autowired public domain MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) - this.amqpAdmin - amqpAdmin; this.amqpTemplate ? amqpTemplate; ? // ... RabbitMessagingTemplate can be injected in a similar way. If a MessageConverter bean is defined, it is automatically associated with the automatically configured AmqpTemplate. If necessary, any org.springframework.amqp.core.Queue that is defined as bean is automatically used to declare a corresponding queue on the RabbitMQ instance. To retry operations, you can enable retries in AmqpTemplate (for example, in case the broker connection is lost): spring.rabbitmq.template.retry.enabled-true spring.rabbitmq.template.retry.initial-interval-2s Reintes are disabled by default. You can also customize the RetryTemplate programmatically by declaring a RabbitRetryTemplateCustomizer bean. If you need to create more rabbitTemplate instances or if you want to override the default, Spring Boot provides a RabbitTemplateConfigurer bean that you can use to initialize a RabbitTemplate with the same settings as the generators used by automatic configuration. When the Rabbit infrastructure is present, any bean can be annotated @RabbitListener to create a listener endpoint. If no RabbitListenerContainerFactory is defined, a SimpleRabbitListenerContainerFactory default value is automatically configured and can be changed to a direct container using the spring.rabbitmq.listener.type property. If a MessageConverter or MessageRecoverer bean is defined, it is automatically associated with the default factory. The following sample component creates a listener endpoint in the someQueue queue: @Component public class MyBean á @RabbitListener(queues ? someQueue) public void processMessage(String content) a // ... See the @EnableRabbit Javadoc for more details. If you need to create more instances or if you want to override the default value, Spring Boot provides a SimpleRabbitListenerContainerFactoryConfigurer and a DirectRabbitListenerContainerFactoryConfigurer that you can use to initialize a SimpleRabbitListenerContainerFactory and a with the same configuration as the factories used by automatic configuration. It doesn't matter what type of container you choose. Those two beans are exposed by automatic configuration. For example, the following configuration class exposes another factory that uses a specific MessageConverter: @Configuration(proxyBeanMethods ? false) static class RabbitConfiguration - @Bean public SimpleRabbitListenerContainerFactory myFactory(SimpleRabbitListenerContainerFactoryConfigurer configurer) Simple ? connectionFactory); factory.setMessageConverter(myMessageConverter()); return factory; You can then use the factory in any @RabbitListener-annotated method, as follows: @Component my public class MyBean ? @RabbitListener (someQueue, containerFactory myFactory) public void processMessage(String content) queues ? You can enable retries to handle situations in which the listener throws an exception. By default, RejectAndDontRequeueRecoverer is used, but you can define your own MessageRecoverer. When retries are exhausted, the message is rejected and discarded or routed to an exchange of unsent messages if the broker is configured to do so. By default, retries are disabled. You can also customize the RetryTemplate programmatically by declaring a RabbitRetryTemplateCustomizer bean. By default, if retries are disabled and the listener throws an exception, delivery is retried indefinitely. You can modify this behavior in two ways: set the defaultRequeueRejected property to false so that zero deliveries are attempted, or throw an AmqpRejectAndDontRequeueException exception to indicate that the message should be rejected. The latter is the mechanism used when retries are enabled and the maximum number of delivery attempts is reached. Apache Kafka supports automatic spring-kafka project configuration. Kafka settings are controlled by external configuration properties in spring.kafka.*. For example, you can declare the following section in application.properties: spring.kafka.bootstrap-servers-localhost:9092 spring.kafka.consumer.group-id-myGroup To create a topic at startup, add a bean of type NewTopic. If the theme already exists, the bean is ignored. See KafkaProperties for more supported options. Spring KafkaTemplate is automatically configured, and you can automatically connect it directly to your own beans, as shown in the following example: @Component public class MyBean to final private KafkaTemplate kafkaTemplate; @Autowired public name MyBean(KafkaTemplate kafkaTemplate) - this.kafkaTemplate - kafkaTemplate; If the spring.kafka.producer.transaction-id-prefix property is set, a KafkaTransactionManager is automatically configured. Also, if you define a bean is automatically associated with the automatically configured KafkaTemplate. When Apache Kafka infrastructure is present, any bean can be annotated with to create a listener endpoint. If No KafkaListenerContainerFactory is defined, a default one is automatically configured with keys defined in spring.kafka.listener.*. The following component creates a listener endpoint in the topic someTopic: @Component public class MyBean ? @KafkaListener(topics ? someTopic) public void processMessage(String content) to // ... If a KafkaTransactionManager bean is defined, it is automatically associated with the container builder. Similarly, if you define an ErrorHandler, AfterRollbackProcessor, or ConsumerAwareRebalanceListener bean, it is automatically associated with the default factory. Depending on the type of listener, a RecordMessageConverter or BatchMessageConverter bean is associated with the default factory. If only one Bean RecordMessageConverter is present for a batch listener, it wraps in a BatchMessageConverter. A custom ChainedKafkaTransactionManager must be marked @Primary as it normally refers to the automatically configured KafkaTransactionManager bean. Spring for Apache Kafka provides a factory bean to create a StreamsBuilder object and manage the lifecycle of its streams. Spring Boot automatically configures the required KafkaStreamsConfiguration bean whenever kafka-streams are in the class path and Kafka Streams is enabled through the @EnableKafkaStreams. Enabling Kafka Streams means that the boot servers and application ID must be set. The first can be configured using spring.kafka.streams.application-id, by default spring.application.name if not set. The latter can be set globally or overridden specifically for sequences only. Several additional properties are available through dedicated properties; other arbitrary Kafka properties can be set using the spring.kafka.streams.properties namespace. See also Additional Kafka Properties for more information. To use the factory bean, connect StreamsBuilder to the @Bean as shown in the following example: @Configuration(proxyBeanMethods - false) @EnableKafkaStreams public static class KafkaStreamsExampleConfiguration - @Bean<Integer, string-> kStream public(StreamsBuilder streamsBuilder) - KStream<Integer stream, string-> - streamBuilders.stream(ks1In); stream.map(k, k, v) -> new KeyValue <> (k, v.toUpperCase())).to(ks1Out, Produced.with(Serdes.Integer(), new JsonSerde <> ())); return stream; By default, streams managed by the StreamBuilder object you create start automatically. You can customize this behavior by using the spring.kafka.streams.auto-startup property. Properties supported by automatic configuration are displayed in Common Application Properties. Keep in mind that, in your these properties (hyphenated or camelCase) map directly to the dotted properties of Apache Kafka. See the Apache Kafka documentation for more information. The first of these properties apply to all components (producers, consumers, administrators, and sequences), but can be specified at the component level if you want to use different values. Apache Kafka designates</Integer,> </Integer,> </Integer,> with the importance of HIGH, MEDIUM or LOW. Spring Boot auto-configuration supports all high-importance properties, some selected MEDIUM and LOW properties, and any properties that do not have a default value. Only a subset of the properties supported by Kafka are available directly through the KafkaProperties class. If you want to configure the producer or consumer with additional properties that are not directly supported, use the following properties: spring.kafka.properties.prop.one-first spring.kafka.admin.properties.prop.two-second spring.kafka.consumer.properties.prop.3-third spring.kafka.producer.properties.prop.four-fourth spring.kafka.streams.properties.prop.five-fifth This sets the common property prop.one Kafka to the first (applies to producers, consumers and administrators), the prop.two admin property second, consumer property prop.three to the third, producer property prop.four to the fourth and prop.five streams property to the fifth. You can also configure Spring Kafka JsonDeserializer as follows: spring.kafka.consumer.value-deserializer.org.springframework.kafka.support.serializer.JsonDeserializer spring.kafka.consumer.properties.spring.json.value.default.type.com.example.Invoice.kafka.consumer.properties.spring.json.trusted.packages.com .com.You can disable JsonSerializer's default behavior of sending type information in headers: spring.kafka.producer.value-serializer.org.springframework.kafka.support.serializer.JsonSerializer spring.kafka.producer.properties.spring.json.add.type.headers-false Properties set in this way override any configuration items that Spring Boot explicitly supports. Spring for Apache Kafka provides a convenient way to test projects with an integrated Apache Kafka broker. To use this feature, note a test class with the @EmbeddedKafka spring-kafka-test module. For more information, see the Spring for Apache Kafka reference manual. For Spring Boot auto-configuration to work with the above-mentioned embedded Apache Kafka broker, you must reassign a system property for embedded broker addresses (populated by EmbeddedKafkaBroker) in the Spring Boot configuration property for Apache Kafka. There are several ways to do this: Provide a system property to assign broker addresses embedded in spring.kafka.bootstrap-servers in the test class: static - System.setProperty(EmbeddedKafkaBroker.BROKER_LIST_PROPERTY, spring.kafka.bootstrap-servers); Configure a property name in the @EmbeddedKafka annotation: @EmbeddedKafka(topics ? someTopic, bootstrapServersProperty ? spring.kafka.bootstrap-servers) Use a placeholder in the configuration properties: If you need to call remote REST services from your application, you can use the Spring Framework RestTemplate class. Because RestTemplate instances often need to be customized before they are used, Spring Boot does not provide any automatic configuration Bean. However, it automatically configures a RestTemplateBuilder, which can be used to instantiate restTemplates when needed. The automatically configured RestTemplateBuilder ensures that sensitive HttpMessageConverters are applied to restTemplate instances. The following code shows a typical example: @Service public class MyService to private final RestTemplate restTemplate; public MyService(RestTemplateBuilder restTemplateBuilder) - this.restTemplate ? restTemplateBuilder.build(); ? public Details someRestCall(String name) ? return this.restTemplate.getForObject(/'name'/details, Details.class, name); RestTemplateBuilder includes a number of useful methods that can be used to quickly configure a RestTemplate. For example, to add auth BASIC support, you can use builder.basicAuthentication(user, password).build(). There are three main approaches to customizing RestTemplate, depending on the wide range of measures in which you want customizations to apply. To make the scope of your customizations as narrow as possible, insert the automatically configured RestTemplateBuilder, and then call its methods as needed. Each method call returns a new instance of RestTemplateBuilder, so customizations only affect this factory usage. To perform application-wide additive customization, use a RestTemplateCustomizer bean. All of these beans are automatically registered with automatically configured RestTemplateBuilder and apply to templates that are compiled with it. The following example shows a customizer that configures the use of a proxy for all hosts except 192.168.0.5: the static ProxyCustomizer class implements RestTemplateCustomizer - @Override public void customize(RestTemplate restTemplate) - HttpHost proxy - new HttpHost(proxy.example.com); HttpClient httpClient to HttpClientBuilder.create().setRoutePlanner(new DefaultProxyRoutePlanner(proxy) to public @Override HttpHost determineProxy(HttpHost target, HttpRequest request, HttpContext context) throws HttpException a if (target.getHostName().equals(192.168.0.5)) - return null; ? return super.determineProxy(target, context; ?). build(); restTemplate.setRequestFactory(new HttpComponentsClientHttpRequestFactory(httpClient)); Finally, the most extreme (and rarely used) option is to create your own RestTemplateBuilder bean. Doing so disables automatic restTemplateBuilder settings and prevents RestTemplateCustomizer beans from being used. If you have Spring WebFlux in the class path, you can also choose to use WebClient to call remote REST services. Compared to RestTemplate, this client has a more functional feel and is fully reactive. You can learn more about WebClient in the Dedicated section of Spring Framework documents. Spring Boot creates and preconfigures a automatically; it is strongly recommended that you inject it into the components and use it to instantiate WebClient. Spring Boot is configuring that constructor to share HTTP resources, reflecting codec settings in the same way as the (see WebFlux's automatic HTTP codec configuration) and more. The following code shows a typical example: @Service public class MyService to private final WebClient webClient; public MyService(WebClient.Builder webClientBuilder) - this.webClient ? webClientBuilder.baseUrl( ).build(); a public Mono<Details> someRestCall(String name) ? return this.webClient.get().uri(/'name'/details, name) .retrieve(). Spring Boot will automatically detect which ClientHttpConnector to use to control WebClient, based on the libraries available in the application class path. For now, Reactor Netty and the Jetty RS client are supported. The spring-boot-starter-webflux starter depends on io.projectreactor.netty:reactor-netty by default, which brings server and client deployments. If you choose to use Jetty as a

reactive server instead, you must add a dependency to the Jetty Reactive HTTP client library, org.eclipse.jetty:jetty-reactive-httpclient. Using the same technology for the server and client has its advantages because it will automatically share HTTP resources between the client and the server. Developers can override resource settings for Jetty and Reactor Netty by providing a Custom ReactorResourceFactory or JettyResourceFactory bean - this will apply to both clients and servers. If you want to override that option for the client, you can define your own ClientHttpConnector bean and have full control over the client configuration. There are three main approaches to customizing WebClient, depending on the wide range of measures in which you want customizations to apply. To make the scope of your customizations as narrow as possible, insert the automatically configured WebClient.Builder and then call its methods as needed. WebClient.Builder instances are stateless: Any changes to the builder are reflected on all clients subsequently created with it. If you want to create multiple clients with the same generator, you can also consider cloning the generator with WebClient.Builder another - builder.clone();. To perform application-wide additive customization for all instances of WebClient.Builder, you can declare WebClientCustomizer beans and change WebClient.Builder locally at the injection point. Finally, you can go back to the original API and use WebClient.create(). In that case, no automatic or WebClientCustomizer settings are applied. The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (such as the Hibernate validator) is in the class path. This allows bean methods to be annotated with javax.validation constraints on their parameters and/or return value. Target classes with these methods they must be annotated with @Validated annotation at the type level for inline constraint annotations to be searched at the type level for their methods to be searched. For example, the following service triggers validation of the first argument, making sure that its size is between 8 and 10: @Service @Validated </Details> </Details> Class MyBean to public file findByCodeAndAuthor(@Size(min to 8, max to 10) String Code, Author Author) . Spring Framework provides an abstraction for sending email using the JavaMailSender interface, and Spring Boot provides automatic configuration for it as well as a startup module. 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 using configuration items in the spring.mail namespace. See MailProperties for details. In particular, certain default timeout values are infinite, and you may want to change it to avoid having a thread blocked by an unresponsive mail server, As shown in the following example: spring.mail.properties.mail.smtp.connectiontimeout-5000 spring.mail.properties.mail.smtp.timeout-3000 spring.mail.properties.mail.smtp.writetimeout-15000 It is also possible to configure a JavaMailSender with an existing session from JNDI: spring.mail.jndi-name-mail/Session When a jndi-name is set, it takes precedence over all other session-related settings. Spring Boot supports JTA transactions distributed across multiple XA resources using an Atomikos embedded transaction manager. Outdated support for using a Bitronix embedded transaction manager is also provided, but will be removed in a future release. JTA transactions are also supported when deploying to an appropriate Java EE application server. When a JTA environment is detected, Spring JtaTransactionManager is used to manage transactions. Automatically configured JMS, DataSource, and JPA beans are updated to support XA transactions. You can use standard spring idioms, such @Transactional, to participate in a distributed transaction. If you are in a JTA environment and still want to use local transactions, you can set the spring.jta.enabled property to false to disable automatic JTA settings. Atomikos is a popular open source transaction manager that can be embedded in the Spring Boot application. You can use the spring-boot-starter-jta-atomikos starter to extract the appropriate Atomikos libraries. Spring Boot automatically configures Atomikos and ensures that the appropriate depend-dependent settings are applied to Spring beans for the correct start and shutdown order. By default, Atomikos transaction logs are written to a transaction log directory in the application's home directory (the directory where the application jar file resides). You can customize the location of this directory by setting a spring.jta.log-dir property in the application.properties file. Properties that begin with spring.jta.atomikos.properties can also be used to customize Atomikos UserTransactionServiceImp. See the Javadoc AtomikosProperties for full details. To ensure that multiple transaction managers can securely coordinate the same resource managers, each Atomikos instance must be configured with a unique identifier. By default, this ID is the IP of the machine atomikos is running on. To ensure uniqueness in production, you must configure the spring.jta.transaction-manager-id property with a different value for each instance of your application. Starting with Spring Boot 2.3, Bitronix support has been deprecated and will be removed in a future release. You can use the spring-boot-starter-jta-bitronix starter to add the appropriate Bitronix dependencies to your project. As with Atomikos, Spring Boot automatically configures Bitronix and then processes its beans to ensure that the start and shutdown order is correct. By default, Bitronix transaction log files (part1.btm and part2.btm) are written to a transaction-logs directory in the application home directory. You can customize the location of this directory by setting the spring.jta.log-dir property. Properties that begin with spring.jta.bitronix.properties are also bound to the bitronix.tm.Configuration bean, allowing for complete customization. See the Bitronix documentation for more information. To ensure that multiple transaction managers can securely coordinate the same resource managers, each Bitronix instance must be configured with a unique identifier. By default, this ID is the IP address of the machine running Bitronix. To ensure uniqueness in production, you must configure the spring.jta.transaction-manager-id property with a different value for each instance of your application. If you package the Spring Boot application as a war or ear file and deploy it to a Java EE application server, you can use the application server's built-in transaction manager. Spring Boot attempts to automatically configure a transaction manager by examining common JNDI locations (java:comp/UserTransaction, java:comp/TransactionManager, and so on). If you are using a transaction service provided by the application server, you generally also want to ensure that all resources are managed by the server and exposed through JNDI. Spring Boot attempts to automatically configure JMS by searching for a ConnectionFactory in the JNDI path (java:/JmsXA or java:/XAConnectionFactory), and you can use the spring.datasource.jndi-name property to configure the dataSource. When using JTA, the primary JMS ConnectionFactory bean supports XA and participates in distributed transactions. In some situations, you may want to process certain JMS messages using a Non-XA ConnectionFactory. For example, JMS processing logic can take longer than the XA timeout. If you want to use a ConnectionFactory other than XA, you can insert the nonXaJmsConnectionFactory bean instead of the jmsConnectionFactory bean @Primary. To maintain the jmsConnectionFactory bean is also provided by the alias of bean xaJmsConnectionFactory. The following example shows how to inject ConnectionFactory instances: // Inject the primary (XA-compatible) ConnectionFactory @Autowired private ConnectionFactory defaultConnectionFactory; Inject the XA ConnectionFactory setting (use the alias and inject the same as above) @Qualifier(xaJmsConnectionFactory) private ConnectionFactory xaConnectionFactory; Inject the connectionfactory @Autowired @Qualifier that is not compatible with the connection @Autowired @Qualifier(nonXaJmsConnectionFactory) private connection from The ConnectionFactory nonXaConnectionFactory; The XAConnectionFactoryWrapper and XADataSourceWrapper interfaces can be used to support alternate embedded transaction managers. Interfaces are responsible for wraping the XAConnectionFactory and XADataSource beans and exposing them as regular ConnectionFactory and DataSource beans, which are transparently enrolled in the distributed transaction. Automatic DataSource and JMS configuration uses JTA variants, as long as you have an appropriate JtaTransactionManager bean and XA container beans registered in your ApplicationContext. If Hazelcast is in the class path and an appropriate configuration is found, Spring Boot automatically configures a HazelcastInstance that you can inject into your application. If you define a com.hazelcast.config.Config bean, Spring Boot uses it. If the configuration defines an instance name, Spring Boot attempts to locate an existing instance instead of creating a new instance. You can also specify the hazelcast configuration file to use through configuration, as shown in the following example: spring.hazelcast.config-classpath:config/my-hazelcast.xml Otherwise Spring Boot attempts to find Hazelcast settings from the default locations: hazelcast.xml in the working directory or at the root of the class path, or a .yaml counterpart in the same locations. We also check if the hazelcast.config system property is set. See the Hazelcast documentation for more details. If hazelcast-client is present in the class path, Spring Boot first attempts to create a client by checking the following configuration options: The presence of a com.hazelcast.client.config.ClientConfig bean. A configuration file defined by the spring.hazelcast.config property. The presence of the hazelcast.client.config system property. A hazelcast-client.xml in the working directory or at the root of the class path. A hazelcast-client.yaml in the working directory or at the root of the class path. Spring Boot offers several amenities to work with the quartz programmer, including the spring-boot-starter-quartz Starter. If Quartz is available, a Scheduler is automatically configured (via the SchedulerFactoryBean abstraction). Beans of the following types are automatically collected and associated with the Scheduler: JobDetail: Defines a particular job. JobDetail instances can be created using the JobBuilder API. Calendar. Trigger: Defines when a particular job is triggered. By default, an in-memory JobStore is used. However, it is possible to set up a warehouse in JDBC if a DataSource bean is available in the application and if the spring.quartz.job-store-type property is configured accordingly, as shown in the following example: spring.quartz.job-store-type-jdbc When using the JDBC store, the schema can be initialized at startup, as shown in the following example: spring.quartz.job-store-type-jdbc When using the JDBC store, the schema can be initialized at startup , as shown in the following example: spring.quartz.job-store-type-jdbc When using the JDBC store, the schema can be initialized at startup, as shown in the following example: spring.quartz.job-job-type-jdbc When using the JDBC store, the schema can be initialized at startup, as shown in the following example: spring.quartz.job-store-type-jdbc When using the JDBC store, the schema can be initialized at startup. , as shown in the following example: spring.quartz.job-job-type-jdbc When using the For example: spring.quartz.jdbc.initialize-schema-always By default, the database is discovered and initialized using the standard scripts provided with the Quartz library. These scripts remove existing tables, eliminating all triggers on each restart. It is also possible to provide a custom script by setting the spring.quartz.jdbc.schema property. To have Quartz use a DataSource other than the application's primary dataSource, declare a DataSource bean, annotating its @Bean with @QuartzDataSource. This ensures that the quartz-specific data source is used by both the SchedulerFactoryBean and for schema initialization. By default, jobs created by the configuration will not overwrite already registered jobs that have been read from a persistent job store. To enable overwriting of existing job definitions, set the spring.quartz.overwrite-existing-jobs property. Quartz Scheduler settings can be customized using the spring.quartz properties and SchedulerFactoryBeanCustomizer beans, which enable programmatic customization of SchedulerFactoryBean. Advanced quartz configuration properties can be customized using spring.quartz.properties.*. In particular, an Executor bean is not associated with the programmer, as Quartz offers a way to configure the scheduler through spring.quartz.properties. If you need to customize the task executor, consider deploying SchedulerFactoryBeanCustomizer. Jobs can define setters to insert data map properties. Regular beans can also be injected similarly, as shown in the following example: the SampleJob public class extends QuartzJobBean - private MyService myService; private String name; Inject MyService bean public void setMyService(MyService myService) to ... ? Inject the job data property name public void setName(String name) ? @Override ... In the absence of an Executor bean in context, Spring Boot automatically configures a ThreadPoolTaskExecutor with reasonable default values that can be automatically associated with asynchronous task execution (@EnableAsync) and Spring MVC asynchronous request processing. If you have defined a custom executor in the context, running regular tasks (that is, @EnableAsync) will use it transparently, but Spring MVC support will not be configured because it requires an implementation of AsyncTaskExecutor (called applicationTaskExecutor). Depending on the target layout, you can change the executor in a ThreadPoolTaskExecutor or define a ThreadPoolTaskExecutor and an AsyncConfigurer to adjust the custom executor. Automatically configured TaskExecutorBuilder allows you to easily create that play what auto-configuration does by default. The thread pool uses 8 main threads that can grow and shrink depending on the load. That default setting can be fine-tuned using the spring.task.execution namespace, as shown in the following example: spring.task.execution.pool.max-size-16 spring.task.execution.pool.max-size-16 spring.task.execution.pool.keep-alive-10s This changes the thread pool to use a delimited queue so that when the queue is full (100 tasks), the thread pool increases to a maximum of 16 threads. Group reduction is more aggressive as threads recover when idle for 10 seconds (was 60 seconds by default). A ThreadPoolTaskScheduler can also be configured automatically if it needs to be associated with the execution of scheduled tasks (@EnableScheduling). The thread pool uses a thread by default, and that setting can be adjusted with the spring.task.scheduling namespace. Both a TaskExecutorBuilder bean and a TaskSchedulerBuilder bean are available in context if you need to create a custom executor or scheduler. Spring Boot offers several amenities to work with Spring Integration, including the spring-boot-starter-integration Starter. Spring Integration provides abstractions about messaging and other transports such as HTTP, TCP, and others. If Spring Integration is available in the class path, it is initialized through the @EnableIntegration. Spring Boot also configures some features that are triggered by the presence of additional spring integration modules. If spring-integration-jmx is also in the class path, message processing statistics are published through JMX. If spring-integration-jdbc is available, the default database schema can be created at startup, as shown in the following line: spring.integration.jdbc.initialize-schema-always If spring-integration-rsocket is available, developers can configure an RSocket server using the spring.rsocket.server.* properties and allow it to use IntegrationRSocketEndpoint or RSocketOutboundGateway components to handle incoming messages. This infrastructure can control spring integration RSocket channel adapters and @MessageMapping drivers (since spring.integration.rsocket.server.message-mapping-enabled is configured). Spring Boot can also automatically configure a ClientRSocketConnector using the configuration properties: ? Connect to an RSocket server over TCP spring.integration.rsocket.client.comotejor.org spring.integration.rsocket.client.port-9898 - Connecting to an RSocket server through WebSocket spring.integration.rsocket.client.uri-ws://example.org By default, if there is a micrometer meterSedy the federation bean, spring integration metrics will be managed by micrometer. If you want to use legacy spring Integration metrics, add a DefaultMetricsFactory bean to the application context. Spring Boot provides automatic Spring Session configuration for a wide range of spring stores When you create a Servlet web application, the following stores can be configured automatically: JDBC Redis Hazelcast MongoDB When you create a reactive web application, the following stores can be configured automatically: If there is a single Spring Session module in the class path, Spring Boot uses that store implementation automatically. If you have more than one deployment, you must choose the StoreType that you want to use to sessions. For example, to use JDBC as a back-end store, you can configure the application as follows: spring.session.store-type-jdbc You can disable Spring Session by setting the store type to none. Each store has specific additional settings. For example, you can customize the table name for the JDBC store, as shown in the following example: spring.session.jdbc.table-name-SESSIONS To set the session timeout, you can use the spring.session.timeout property. If that property is not set, the automatic setting returns to the value of server.servlet.session.timeout. Java Management Extensions (JMX) provides a standard mechanism for monitoring and managing applications. Spring Boot exposes the most suitable MBeanServer as a bean with an ID of mbeanServer. Any of your beans that are annotated with spring JMX annotations (@ManagedResource, @ManagedAttribute, or @ManagedOperation) are exposed to it. If the platform provides a standard MBeanServer, Spring Boot will use it and use it by default for the MBeanServer virtual machine if necessary. If all of that fails, a new MBeanServer will be created. See the JmxAutoConfiguration class for details. Spring Boot provides a number of utilities and annotations to help you test your application. Test support is provided by two modules: spring-boot-test contains main elements, and spring-boot-test-autoconfigure supports automatic configuration for testing. Most developers use the Starter spring boot test, which imports both spring boot and JUnit Jupiter, AssertJ, Hamcrest, and a number of other useful libraries. The starter also brings the vintage engine so you can run the JUnit 4 and JUnit 5 tests. If you migrated the tests to JUnit 5, You should exclude JUnit 4 support, as shown in the following example: <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-boot-starter-test</artifactId> <scope>test</scope> <exclusions> <exclusions> <exclusions> <exclusions> <exclusions> <exclusion> <groupId>org.junit.vintage</groupId> <artifactId>junit-vintage junit-vintage-engine</artifactId> </exclusion> </exclusions> </dependency> The spring-boot-starter-test Starter (at the test scope) contains the following libraries provided: JUnit 5 (including the vintage engine for backward compatibility with JUnit 4): The de facto standard for unit test Java applications. Spring Test and Spring Boot Test: Utilities and integration test support for spring Boot applications. AssertJ: A fluid assertion library. Hamcrest: A library of matching objects (also known as constraints or predicates). Mockito: A simulation framework JSONassert: An assertion library for JSON. JsonPath: XPath for JSON. These common libraries are generally useful to us when writing tests. If these libraries don't suit your needs, you can add additional test dependencies of your own. One of the main advantages of dependency injection is that you should make it easier to unit test your code. You can instantiate objects using the new operator without that involves spring. You can also use dummy objects instead of actual dependencies. Often, you must go beyond unit tests and start integration tests (with Spring ApplicationContext). It is useful to be able to perform integration tests without the need to deploy the application or need to connect to another infrastructure. Spring Framework includes a dedicated test module for such integration tests. You can declare a dependency directly on org.springframework:spring-test or use spring-boot-starter-test Starter to extract it transitively. If you have not used the spring-test module before, you should start by reading the corresponding section of the Spring Framework reference documentation. A Spring Boot application is a Spring ApplicationContext, so you don't have to do anything very special to test it beyond what you would normally do with a vanilla spring context. Spring Boot external properties, logging, and other features are installed in context by default only if you use SpringApplication to create it. If you are using JUnit 4, be sure to also add @RunWith(SpringRunner.class) to your test, otherwise annotations will be ignored. If you are using JUnit 5, there is no need to add the equivalent @ExtendWith(SpringExtension.class) as @SpringBootTest and the other ... Test annotations are already annotated with it. By default, the @SpringBootTest will not start a server. You can use the @SpringBootTest webEnvironment attribute to further refine how tests run: MOCK(Default): Loads a web ApplicationContext and provides a dummy web environment. Embedded servers do not start when you use this annotation. If a web environment is not available in the class path, this mode transparently returns to creating a normal non-web ApplicationContext. It can be used @AutoConfigureMockMvc or @AutoConfigureWebTestClient for simulation-based testing of the web application. RANDOM_PORT: Loads a WebServerApplicationContext and provides a real web environment. Embedded servers start and listen on a random port. DEFINED_PORT: Loads a WebServerApplicationContext and provides a real web environment. Embedded servers start and listen on a defined port (from their application.properties) or on the default port of 8080. NONE: Loads an ApplicationContext using SpringApplication but does not provide any web environments (mock or other). If the test is @Transactional, roll back the transaction at the end of each test method by default. However, because using this provision with RANDOM_PORT or DEFINED_PORT implicitly provides an actual servlet environment, the HTTP client and server run on separate threads and therefore in separate transactions. Any transaction on the server is not rolled back in this case. @SpringBootTest with webEnvironment - WebEnvironment.RANDOM_PORT will also start the management server on a separate random port if the application uses a different port for management If Spring MVC is available, a normal MVC-based application context is configured. If you only have Spring WebFlux, we'll detect it and configure a WebFlux-based application context instead. If both are present, Spring MVC takes precedence. If you want to test a reactive web application in this scenario, you must set the spring.main.web-application-type: @SpringBootTest(properties ? spring.main.web-application-type-reactive) class to MyWebFluxTests . If you are familiar with spring test framework, you may be used to using @ContextConfiguration(classes-...)) to specify which @Configuration Spring to load. Alternatively, you may have often used classes of @Configuration nested within the test. When testing Spring Boot applications, this is often not necessary. Spring Boot test annotations automatically search for their primary settings each time one is not explicitly defined. The search algorithm works from the package that contains the test until it finds a class annotated with @SpringBootApplication or @SpringBootConfiguration. As long as you have structured the code in a sensible way, you usually find the main configuration. The underlying component analysis settings in @SpringBootApplication defines the exclusion filters that are used to ensure that the cut works as expected. If you use an explicit @ComponentScan in the @SpringBootApplication class, note that those filters will be disabled. If you are using the cut, you must define them again. If you want to customize the primary settings, you can use a nested @TestConfiguration class. Unlike a nested @Configuration class, which would be used instead of the main application configuration, a nested @TestConfiguration class is used in addition to the main application settings. Spring's test framework caches application contexts between tests. Therefore, whenever tests share the same configuration (regardless of how it is detected), the potentially slow process of loading the context occurs only once. If your application uses component analysis (for example, if you are using @SpringBootApplication or @ComponentScan), you might encounter top-level configuration classes that you created only for specific tests that are accidentally collected everywhere. As we've seen earlier, @TestConfiguration can be used in an internal class of a test to customize the primary configuration. When placed in a higher-level class, @TestConfiguration indicates that classes in src/test/java should not be collected by scanning. You can then explicitly import that class where needed, as shown in the following example: @SpringBootTest Class MyTests - @Test void exampleTest() to ... If you use @ComponentScan (that is, not through @SpringBootApplication) you must register the TypeExcludeFilter with it. See Javadoc for more information. If your application expects arguments, you can inject them using the args attribute. @SpringBootTest(args a --app.test-one) Class ApplicationArgumentsExampleTests - @Test void applicationArgumentsPopulated(@Autowired ApplicationArguments args) ? assertThat(args.getOptionNames()).containsOnly(app.test); assertThat(args.getOptionValues(app.test)). Contains(one); By default, the @SpringBootTest does not start the server. If you have web endpoints that you want to test with this dummy environment, you can also configure MockMvc as shown in the following example: import org.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.web.servlet.MockMvc; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status; @SpringBootTest @AutoConfigureMockMvc MockMvcExampleTests a @Test void exampleTest(@Autowired MockMvc mvc) class produces Exception a mvc.perform(get(/)).andExpect(status().isOk()).andExpect(content().string(Hello World)); Alternatively, you can configure a WebTestClient as shown in the following example: import org.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.web.reactive.server.WebTestClient; @SpringBootTest @AutoConfigureWebTestClient MockWebTestClientExampleTests a @Test void exampleTest(@Autowired WebTestClient webClient) class to webClient.get().uri(/).exchange().expectStatus().isOk().expectBody(String.class).isEqualTo(Hello World); Testing in a simulated environment is typically faster than running with a full Servlet container. However, because simulation occurs in the Spring MVC layer, code that is based on the behavior of the lower-level Servlet container cannot be tested directly with MockMvc. For example, Spring Boot error handling is based on the error page support provided by the Servlet container. This means that while you can test MVC layer launches and handle exceptions as expected, you cannot directly prove that a specific custom error page is rendered. If you need to test these lower-level concerns, you can start a fully running server as described in the next section. If you need to start a fully running server, we recommend that you use random ports. If you @SpringBootTest(webEnvironment-WebEnvironment.RANDOM_PORT), a port is selected randomly every time the test is run. The annotation @LocalServerPort can be used to inject the actual port used in the test. For convenience, the tests you need the started server can also @Autowire a WebTestClient, which resolves links to the running server and comes with @Autowire dedicated API to verify responses, as shown in the following example: import org.junit.jupiter.api.Test; 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.test.web.reactive.server.WebTestClient; @SpringBootTest(webEnvironment a WebEnvironment.RANDOM_PORT) RandomPortWebTestClientExampleTests class? @Test void exampleTest(@Autowired WebTestClient webClient) to webClient.get().uri(/).exchange().expectStatus().isOk().expectBody(String.class).isEqualTo Hello(World); This configuration requires spring-webflux in the class path. If you cannot or will not add webflux, Spring Boot also provides a TestRestTemplate installation: import org.junit.jupiter.api.Test; 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 static org.assertj.core.api.Assertions.assertThat; @SpringBootTest(webEnvironment ? WebEnvironment.RANDOM_PORT) RandomPortTestRestTemplateExampleTests class? @Test void exampleTest(@Autowired TestRestTemplate restTemplate) ? To customize the WebTestClient bean, configure a WebTestClientBuilderCustomizer bean. All of these beans are called with the WebTestClient.Builder that is used to create the WebTestClient. As the test context framework caches the context, JMX is disabled by default to prevent identical components from registering in the same domain. If such a test needs access to an MBeanServer, consider marking it dirty too: @ExtendWith(SpringExtension.class) @SpringBootTest(properties ? spring.jmx.enabled-true) @DirtiesContext sampleJmxTests - @Autowired MBeanServer mBeanServer private; @Test void exampleTest() to // á . When running tests, it is sometimes necessary to simulate certain components within the context of the application. For example, you might have a facade over some remote service that is not available during development. Simulation can also be useful when you want to simulate errors that might be difficult to trigger in a real-world environment. Spring Boot includes a @MockBean that can be used to define a mockite mock for a bean within its ApplicationContext. You can use annotation to add new beans or replace a single existing bean definition. Annotation can be used directly in test classes, in fields within the test, or in classes and fields @Configuration. When used in a field, the instance of the created drill is also injected. The are automatically reset after each test method. If the test uses use Spring Boot test annotations (such as @SpringBootTest), this feature is automatically enabled. To use this feature with a different layout, you must explicitly add a listener, as shown in the following example: @TestExecutionListeners(MockitoTestExecutionListener.class) The following example replaces an existing RemoteService bean with a simulated implementation: import.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.*; import org.springframework.boot.test.context.*; import org.springframework.boot.test.mock.mockito.*; import static org.assertj.core.api.Assertions.*; import static org.mockito.BDDMockito.*; @SpringBootTest MyTests class to @MockBean Private RemoteService remoteService; @Autowired back private Reverser; @Test void exampleTest() to // RemoteService has been injected into the reverse bean given(this.remoteService.someCall()).willReturn(mock); String reverse - reverser.reverseSomeCall(); assertThat(reverse).isEqualTo(kcom); You cannot use @MockBean to simulate the behavior of a bean that is exercised during application context update. At the time the test runs, the application context update is complete and is too late to configure simulated behavior. We recommend that you use a @Bean method to create and configure the drill in this situation. In addition, you @SpyBean to wrap any existing bean with a Mockito spy. See Javadoc for more information. CGLib proxies, such as those created for scoped beans, declare proxy methods as final. This prevents Mockito from working properly, as it cannot circumvent or spy on the final methods in its default settings. If you want to mock or spy on such a bean, configure Mockito to use your online drill manufacturer by adding org.mockito:mockito-inline to the app's test dependencies. This allows Mockito to mock and spy on the final methods. While spring's test framework caches application contexts between tests and reuses a context for tests that share the same configuration, using @MockBean or @SpyBean influences the cache key, most likely increasing the number of contexts. If you @SpyBean to spy on a bean with @Cacheable methods that reference parameters by name, the application must be compiled with -parameters. This ensures that parameter names are available to the caching infrastructure after the bean has been spied on. When using @SpyBean to spy on a bean that is proxied by Spring, you may need to remove spring proxy in some situations, for example, by setting expectations using given or when. Use AopTestUtils.getTargetObject(yourProxiedSpy) to do so. The Spring Boot's automatic setup works well for applications, but sometimes it can be too much for testing. It often helps load only the parts of the configuration needed to test an application slice. For example, you might want to prove that Spring MVC controllers are assigning URLs correctly and you don't want to involve calls in those tests, or you might want to test JPA features, and you're not interested in the web layer when those tests run. The spring-boot-test-autoconfigure module includes a series 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 ApplicationContext and one or more @AutoConfigure... annotations that can be used to customize automatic settings. Each slice restricts component analysis to the appropriate components and loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most of them... Test annotations provide an excludeAutoConfiguration attribute. Alternatively, you can use @ImportAutoConfiguration-exclude. Including multiple slices by using several ... Test annotations are not supported in a test. If you need multiple slices, choose one of the ... Test the annotations and include the @AutoConfigure... annotations of the other slices by hand. It is also possible to use the @AutoConfigure... annotations with standard annotation @SpringBootTest. You can use this combination if you are not interested in cutting the app, but you want some of the test beans set up automatically. To prove that the object's JSON serialization and deserialization works as expected, you can use the @JsonTest. @JsonTest configures the available supported JSON mapper, which can be one of the following libraries: Jackson ObjectMapper, any @JsonComponent beans, and any Jackson Gson Jsonb module If you need to configure auto-configuration elements, you can use @AutoConfigureJsonTesters annotation. Spring Boot includes AssertJ-based helpers that work with the JSONAssert and JsonPath libraries to verify that JSON appears as expected. The JacksonTester, GsonTester, JsonbTester, and BasicJsonTester classes can be used for Jackson, Gson, Jsonb, and Strings respectively. Auxiliary fields in the test class can be @Autowired when using @JsonTest. The following example shows a test class for Jackson: import org.junit.jupiter.api.Test; 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 static org.assertj.core.api.Assertions.*; @JsonTest JacksonTester<VehicleDetail @Autowired s> json; @Test void testSerialize() launches Exception - VehicleDetails details - new VehicleDetails(Honda, Civic); Assert against a '.json' file in the same package as the assertThat(this.json.write(details)) test).isEqualToJson(expected.json); Or use JSON path-based assertions // O utilizar aserciones basadas en rutas JSON assertThatThat(this.json.write(details))).hasJsonPathStringValue(-.make); assertThat(this.json.write(details)).extractingJsonPathStringValue(-.make) .isEqualTo(Honda); @Test ? </VehicleDetails> </VehicleDetails> AssertThat(this.json.parse(content)) .isEqualTo(new VehicleDetails(Ford, Focus)); AssertThat(this.json.parseObject(content).getMake()).isEqualTo(Ford); JSON helper classes can also be used directly in standard unit tests. To do this, call the helper's initFields method on the @Before if you do not use @JsonTest. If you use Spring Boot AssertJ-based helpers to assert a numeric value in a particular JSON path, you might not be able to use isEqualTo based on the type. Instead, you can use AssertJ satisfactions to assert that the value matches the given condition. For example, the following example states that the actual number is a float value close to 0.15 within an offset of 0.01. AssertThat(json.write(message)) .extractingJsonPathNumberValue(.test.numberValue) .satisfies((number) -> assertThat(number.floatValue()).isCloseTo(0.15f, within(0.01f))); To test whether Spring MVC controllers work as expected, use the @WebMvcTest. @WebMvcTest automatically configures the Spring MVC infrastructure and limits scanned beans to @Controller, @ControllerAdvice, @JsonComponent, Converter, GenericConverter, Filter, HandlerInterceptor, WebMvcConfigurer, and HandlerMethodArgumentResolver. Regular @Component and @ConfigurationProperties beans are not scanned when using the @WebMvcTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. If you need to register additional components, such as the Jackson module, you can import additional configuration classes @Import in the test. Often, @WebMvcTest is limited to a single controller and is used in combination with @MockBean to provide simulated implementations for the required collaborators. @WebMvcTest also automatically configures MockMvc. Mock MVC provides an effective way to quickly test MVC controllers without starting a full HTTP server. You can also automatically configure MockMvc in a @WebMvcTest (for example, @SpringBootTest) by noting it with @AutoConfigureMockMvc. The following example uses MockMvc: import org.junit.jupiter.api.*; 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.*; @WebMvcTest(UserVehicleController.class) class MyControllerTests - @Autowired private mockMvc mvc; @MockBean userVehicleService userVehicleService private account; @Test void testExample() produces Exception (this.userVehicleService.getVehicleDetails(sboot)) .willReturn(new Civic)); this.mvc.perform(get(/sboot/vehicle).accept(MediaType.TEXT_PLAIN)) .andExpect(status().isOk()).andExpect(content(). If you need to configure items automatic configuration (for example, when servlet filters should be applied) can use attributes in the @AutoConfigureMockMvc. If you use HtmlUnit or Selenium, automatic configuration also provides an HtmlUnit WebClient bean and/or a Selenium WebDriver bean. The following example uses HtmlUnit: import com.gargoylesoftware.htmlunit.*; import org.junit.jupiter.api.*; 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.*; @WebMvcTest(UserVehicleController.class) class MyHtmlUnitTests - @Autowired webClient webClient; @MockBean userVehicleService userVehicleService private account; @Test void testExample() throws Exception ? given(this.userVehicleService.getVehicleDetails(sboot)) .willReturn(new VehicleDetails(Honda, Civic)); Html page: this.webClient.getPage(/sboot/vehicle.html); AssertThat(page.getBody().getTextContent()).isEqualTo(Honda Civic); By default, Spring Boot places WebDriver beans in a special scope to ensure that the driver exits after each test and that a new instance is injected. If you do not want this behavior, you @Scope(singleton) to the WebDriver @Bean definition. The webDriver scope created by Spring Boot will override any user-defined scope of the same name. If you define your own webDriver scope, it may stop working when you use @WebMvcTest. If you have Spring Security in the class path, @WebMvcTest will also parse the WebSecurityConfigurer beans. Instead of completely disabling security for such tests, you can use Spring Security test support. More details on how to use Spring Security MockMvc support can be found in this Testing With Spring Security procedures section. To prove that Spring WebFlux drivers work as expected, you can use the @WebFluxTest. @WebFluxTest automatically configures spring WebFlux infrastructure and limits scanned beans to @Controller, @ControllerAdvice, @JsonComponent, Converter, GenericConverter, WebFilter, and WebFluxConfigurer. Regular @Component and @ConfigurationProperties beans are not scanned when using the @WebFluxTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. If you need to register additional components, such as the Jackson Module, you can import additional configuration classes @Import in the test. Often, @WebFluxTest limited to a single controller and is used in combination with @MockBean annotation to provide simulated implementations for the required collaborators. @WebFluxTest also automatically configures WebTestClient, which a powerful way to quickly test WebFlux drivers without starting a full HTTP server. You can also automatically configure WebTestClient on a @WebFluxTest (for example, @SpringBootTest) by noting it with @AutoConfigureWebTestClient. The following displays a class that uses @WebFluxTest and WebTestClient: import org.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest; import org.springframework.http.MediaType; import org.springframework.test.web.reactive.server.WebTestClient; @WebFluxTest(UserVehicleController.class) class MyControllerTests - @Autowired WebTestClient webClient; @MockBean userVehicleService userVehicleService private account; @Test void testExample() produces Exception . Civic)); this.webClient.get().uri(/sboot/vehicle).accept(MediaType.TEXT_PLAIN) .exchange() .expectStatus().isOk() .expectBody(String.class).isEqualTo(Honda Civic); This setting is only supported for WebFlux applications, because using WebTestClient in a simulated web application only works with WebFlux at this time. @WebFluxTest cannot detect routes registered through the functional web framework. To test RouterFunction beans in context, consider importing your RouterFunction yourself through @Import or using @SpringBootTest. @WebFluxTest cannot detect a custom security setting registered through a @Bean of type SecurityWebFilterChain. To include it in the test, you must import the configuration that registers the bean through @Import or use @SpringBootTest. You can use @DataJpaTest annotation to test JPA applications. By default, it searches for @Entity classes and configures JPA repositories for spring data. If an embedded database is available in the class path, it also configures one. Regular @Component and @ConfigurationProperties are not analyzed when using the @DataJpaTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. By default, JPA data tests are transactional and rolled back at the end of each test. See the appropriate section in the Spring Framework reference documentation for more details. If that's not what you want, you can disable transaction management for a test or for the entire class as follows: import org.junit.jupiter.api.Test; import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @DataJpaTest @Transactional(propagation - Propagation.NOT_SUPPORTED) Class ExampleNonTransactionalTests - - JPA data tests can also inject a TestEntityManager bean, which provides an alternative to the standard EntityManager JPA that is designed specifically for testing. If you want to use TestEntityManager @DataJpaTest instances, you can also use the @AutoConfigureTestEntityManager. A JdbcTemplate is also available if you need to. In the the following is the annotation @DataJpaTest in use: import org.junit.jupiter.api.Test; Import Import import static org.assertj.core.api.Assertions.*; @DataJpaTest ExampleRepositoryTests class to @Autowired private TestEntityManager entity; @Autowired private UserRepository repository; @Test void testExample() produces Exception ? this.entityManager.persist(new User(sboot, 1234)); User: this.repository.findByUsername(sboot); AssertThat(user.getUsername()).isEqualTo(sboot); assertThat(user.getVin()).isEqualTo(1234); • In-memory-embedded databases generally work well for testing, as they are fast and require no installation. However, if you prefer to run tests on an actual database, you can use the @AutoConfigureTestDatabase annotation, as shown in the following example: @DataJpaTest @AutoConfigureTestDatabase(replace-Replace.NONE) Class ExampleRepositoryTests - // ... @JdbcTest is similar to @DataJpaTest but is for tests that require only one DataSource and do not use Spring Data JDBC. By default, it configures an in-memory embedded database and a JdbcTemplate. Regular @Component and @ConfigurationProperties are not analyzed when using the @JdbcTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. By default, JDBC tests are transactional and rolled back at the end of each test. See the appropriate section in the Spring Framework reference documentation for more details. If that's not what you want, you can disable transaction management for a test or for the entire class, as follows: import org.junit.jupiter.api.Test; import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @JdbcTest @Transactional(propagation - Propagation.NOT_SUPPORTED) Class ExampleNonTransactionalTests - If you prefer the test to run on an actual database, you can use the @AutoConfigureTestDatabase annotation in the same way as for DataJpaTest. (See JPA testing of automatically configured data.) @DataJdbcTest is similar to @JdbcTest but is for tests that use Spring Data JDBC repositories. By default, it configures an in-memory embedded database, a JdbcTemplate, and Spring Data JDBC repositories. Regular @Component and @ConfigurationProperties are not analyzed when using the @DataJdbcTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. By default, JDBC data tests are transactional and rolled back at the end of each test. See the appropriate section in the Spring Framework reference documentation for more details. If that's not what you want, you can disable transaction management for a test or for the entire test class shown in the JDBC sample. If you prefer the test to run on an actual database, you can use the @AutoConfigureTestDatabase in the same way as for DataJpaTest. (See JPA testing of automatically configured data.) You can @JooqTest similarly to @JdbcTest but for jOOQ-related tests. Como Como largely depends on a Java-based schema that corresponds to the database schema, the existing dataSource is used. If you want to replace it with an in-memory database, you can use @AutoConfigureTestDatabase to override that setting. (For more information about using jOOQ with Spring Boot, see Using jOOQ earlier in this chapter.) Regular @Component and @ConfigurationProperties are not analyzed when using the @JooqTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. @JooqTest configures a DSLContext. The following example shows the annotation for @JooqTest in use: import org.jooq.DSLContext; import org.junit.jupiter.api.Test; import org.springframework.boot.test.autoconfigure.jooq.JooqTest; @JooqTest examplejooqTests class @Autowired DSLContext dslContext private; JOOQ tests are transactional and rolled back at the end of each test by default. If that's not what you want, you can disable transaction management for a test or for the entire test class as shown in the JDBC sample. You can @DataMongoTest to test MongoDB applications. By default, you configure an in-memory-embedded MongoDB (if available), configure a MongoTemplate, search for @Document classes, and configure Spring Data MongoDB repositories. Regular @Component and @ConfigurationProperties are not analyzed when using the @DataMongoTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. (For more information about using MongoDB with Spring Boot, see MongoDB, earlier in this chapter.) A list of automatic configuration settings enabled by the application can be found @DataMongoTest. The following class shows the annotation @DataMongoTest in use: import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest; import org.springframework.data.mongodb.core.MongoTemplate; @DataMongoTest ExampleDataMongoTests class - @Autowired MongoTemplate private MongoTemplate;

- In embedded mongoDB memory usually works well for testing as it is fast and does not require any developer installation. However, if you prefer to run tests on an actual MongoDB server, you must exclude the automatic configuration of embedded MongoDB, as shown in the following example: import org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration; import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest; @DataMongoTest(excludeAutoConfiguration - EmbeddedMongoAutoConfiguration.class) class ExampleDataMongoNonEmbeddedTests @DataNeo4jTest . By default, it uses an in-memory-embedded Neo4j (if the embedded driver is searches for classes @NodeEntity and configures Spring Data Neo4j repositories. Regular @Component and @ConfigurationProperties are not analyzed when using the @DataNeo4jTest. @EnableConfigurationProperties can be used to @ConfigurationProperties beans. (For more information on using Neo4J with Spring Boot, see Neo4j, earlier in this chapter.) A list of automatic configuration settings enabled by the application can be found in @DataNeo4jTest. The following example shows a typical configuration for using Neo4J tests in Spring Boot: import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest; @DataNeo4jTest ExampleDataNeo4jTests class - @Autowired private repository YourRepository; - By default, Data Neo4j tests are transactional and rolled back at the end of each test. See the appropriate section in the Spring Framework reference documentation for more details. If that's not what you want, you can disable transaction management for a test or for the entire class, as follows: import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest; import org.springframework.transaction.annotation.Propagation; import org.springframework.transaction.annotation.Transactional; @DataNeo4jTest @Transactional(propagation - Propagation.NOT_SUPPORTED) Class ExampleNonTransactionalTests - - You can use @DataRedisTest to test Redis applications. By default, it searches for @RedisHash classes and configures Spring Data Redis repositories. Regular @Component and @ConfigurationProperties beans are not scanned when using the @DataRedisTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. (For more information about using Redis with Spring Boot, see Redis, earlier in this chapter.) A list of automatic configuration settings enabled by the application can be found @DataRedisTest. The following example shows the annotation for @DataRedisTest in use: import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest; @DataRedisTest ExampleDataRedisTests class - @Autowired yourRepository private repository; - You can @DataLdapTest to test LDAP applications. By default, you configure an in-memory LDAP (if available), configure an LdapTemplate, search for @Entry, and configure Spring Data LDAP repositories. Regular @Component and @ConfigurationProperties are not analyzed when using the @DataLdapTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. (For more information about using LDAP with Spring Boot, see LDAP earlier in this chapter.) A list of automatic configuration settings enabled by the application can be found @DataLdapTest. The following example shows the annotation @DataLdapTest in import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest; import org.springframework.ldap.core.LdapTemplate; @DataLdapTest ExampleDataLdapTests class - @Autowired LDAPTemplate ldapTemplate private; - In-memory LDAP generally works well for as it is fast and does not require any installation from the developer. However, if you prefer to run tests on an actual LDAP server, you must exclude the embedded LDAP automatic configuration, as shown in the following example: import org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration; import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest; @DataLdapTest(excludeAutoConfiguration - EmbeddedLdapAutoConfiguration.class) class ExampleDataLdapNonEmbeddedTests - - You can use the @RestClientTest annotation to test REST clients. By default, it automatically configures support for Jackson, GSON, and Jsonb, configures a RestTemplateBuilder, and adds support for MockRestServiceServer. Regular @Component and @ConfigurationProperties are not analyzed when using the @RestClientTest. @EnableConfigurationProperties can be used to include @ConfigurationProperties. A list of automatic configuration settings enabled by the application can be found in @RestClientTest. The specific beans you want to test must be specified using the value attribute or @RestClientTest components, as shown in the following example: @RestClientTest(RemoteVehicleDetailsService.class) Class ExampleRestClientTest - @Autowired Private Service RemoteVehicleDetailsService; @Autowired mockRestServiceServer private server; @Test void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails() produces Exception to this.server.expect(requestTo(/greet/details)) .andRespond(withSuccess(hello, MediaType.TEXT_PLAIN)); String greeting: this.service.callRestService(); assertThat(greeting).isEqualTo(hello); • You can use @AutoConfigureRestDocs annotation to use Spring REST Documents in tests with Mock MVC, REST Assured, or WebTestClient. Eliminates the need for the JUnit extension in Spring REST Docs. @AutoConfigureRestDocs can be used to override the default output directory (target/generated-snippets if you are using Maven or build/generated-snippets if you are using Gradle). It can also be used to configure the host, schema, and port that appears in documented URIs. @AutoConfigureRestDocs the MockMvc bean to use Spring REST Docs when testing Servlet-based web applications. You can inject it @Autowired and use it in tests as you normally would when using Mock MVC and Spring REST Docs, as shown in the following example: import org.junit.jupiter.api.Test; 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.web.servlet.MockMvc; import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document; import static import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*; @WebMvcTest(UserController.class) @AutoConfigureRestDocs @AutoConfigureRestDocs UserDocumentationTests a @Autowired mockMvc mvc private pool; @Test void listUsers() throws Exception to this.mvc.perform(get(/users).accept(MediaType.TEXT_PLAIN)) .andExpect(status().isOk()) .andDo(document(list-users)); If you need more control over spring REST Docs settings than is offered by @AutoConfigureRestDocs attributes, you can use a RestDocsMockMvcConfigurationCustomizer bean, as shown in the following example: @TestConfiguration Static Class CustomizationConfiguration implements RestDocsMockMvcConfigurationCustomizer ? @Override public void customize(MockMvcRestDocumentationConfigurer configurer) ? configurer.snippets().with. Template.. If you want to make use of Spring REST Docs support for a parameterized output directory, you can create a RestDocumentationResultHandler bean. AlwaysDo auto-configuration calls with this result handler, which causes each call to MockMvc to automatically generate the default code snippets. The following example shows a RestDocumentationResultHandler that is being defined: @TestConfiguration(proxyBeanMethods) static class ResultHandlerConfiguration , @Bean restDocumentationResultHandler public restDocumentation() , return MockMvcRestDocumentation.document(-method-name-@AutoConfigureRestDocs); You can inject it using @Autowired and use it in tests as you normally would when using @WebFluxTest and Spring REST Docs, as shown in the following example: import org.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs; import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest; import org.springframework.test.web.reactive.server.WebTestClient; import static org.springframework.restdocs.webtestclient.WebTestClientRestDocumentation.document; @WebFluxTest @AutoConfigureRestDocs UsersDocumentationTests class to @Autowired private WebTestClient webTestClient; @Test void listUsers() to this.webTestClient.get().uri(/).exchange().expectStatus().isOk().expectBody() .consumeWith(document(list-users)); If you need more control over spring REST Docs settings than is offered by @AutoConfigureRestDocs attributes, you can use a RestDocsWebTestClientConfigurationCustomizer bean, as shown in the following example: @TestConfiguration(proxyBeanMethods ? false) public static class CustomizationConfiguration implements RestDocsWebClientTestConfigurationCustomizer ? @Override public void customize(WebTestClientRestDocumentationConfigurer configurer) ? configurer.snippets().withEncoding(UTF-8@AutoConfigureRestDocs); You can inject it @Autowired and use it in your tests as you normally would when using REST Assured and Spring REST Docs, the following example shows: import io.restassured.specification.RequestSpecification; import org.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.boot.test.context.SpringBootTest.WebEnvironment; import org.springframework.boot.web.server.LocalServerPort; import static io.restassured.RestAssured.given; import org.hamcrest.Matchers.is functions; import static org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document; @SpringBootTest(webEnvironment a WebEnvironment.RANDOM_PORT) @AutoConfigureRestDocs UserDocumentationTests class? @Test void listUsers(@Autowired RequestSpecification documentationSpec, @LocalServerPort int port) á given(documentationSpec).filter(document(list-users)).when().port(port).get(/).then().assertThat() .statusCode(is(200)); If you need more control over spring REST Docs settings than is offered by @AutoConfigureRestDocs attributes, you can use a RestDocsRestAssuredConfigurationCustomizer bean, as shown in the following example: @TestConfiguration(proxyBeanMethods) public static class CustomizationConfiguration implements RestDocsRestAssuredConfigurationCustomizer ? @Override public void customize(RestAssuredRestDocumentationConfigurer configurer) ? configurer.snippets().withTemplateFormat(TemplateFormats.markdown()); You can @WebServiceClientTest to test applications that use call web services by using the Spring Web Services project. By default, it configures a dummy WebServiceServer bean and automatically customizes WebServiceTemplateBuilder. (For more information about using web services with Spring Boot, see Web Services earlier in this chapter.) A list of the automatic configuration settings enabled by the application can be found @WebServiceClientTest. The following example shows the @WebServiceClientTest annotation in use: @WebServiceClientTest(ExampleWebServiceClient.class) class WebServiceClientIntegrationTests - @Autowired mockWebServiceServer server; @Autowired exampleWebServiceClient private client; @Test void mockServerCall() to this.server.expect(payload(new StringSource(<request></request>))).andRespond(withPayload(new StringSource(<response><status>200</status></response>))); assertThat(this.client.test()).extracting(Response::getStatus).isEqualTo(200); Each slice provides one or more @AutoConfigure... annotations that define the automatic configurations to include as part of a slice. Additional automatic configurations can be added test-by-test by creating a @AutoConfigure application... annotation or adding @ImportAutoConfiguration to the test as it is in the following example: @JdbcTest @ImportAutoConfiguration(IntegrationAutoConfiguration.class) Class ExampleJdbcTests - - Make sure that you do not use the @Import @Import to import automatic configurations as they are handled in a specific way by Spring Boot. Alternatively, additional automatic configurations can be added for any use of an industry annotation by registering them in META-INF/spring.factories as shown in the following example: org.springframework.boot.test.autoconfigure.jdbc.JdbcTest-com.example.IntegrationAutoConfiguration A slice or @AutoConfigure... annotation can be customized this way as long as it is meta-annotated with @ImportAutoConfiguration. It then becomes important not to dirty the main class of the application with configuration settings that are specific to a particular area of its functionality. Suppose you are using Spring Batch and trust the automatic configuration for it. You can define your @SpringBootApplication as follows: @SpringBootApplication @EnableBatchProcessing SampleApplication public class . . . . Because this class is the source configuration for the test, any industry test actually tries to start Spring Batch, which is definitely not what you want to do. A recommended approach is to move that area-specific configuration to a separate @Configuration class at the same level as the application, as shown in the following example: @Configuration(proxyBeanMethods - false) @EnableBatchProcessing batchConfiguration public class. Depending on the complexity of your application, you can have a single @Configuration for customizations or one class per domain area. This latter approach allows you to enable it in one of the tests, if necessary, with the @Import. Test slices exclude @Configuration of exploration classes. For example, for a @WebMvcTest, the following configuration will not include the given WebMvcConfigurer bean in the context of the application loaded by the test sector: @Configuration WebConfiguration public class @Bean . However, the following configuration will cause the test industry to load the custom WebMvcConfigurer. @Component TestWebMvcConfigurer implements WebMvcConfigurer . . . . Another source of confusion is class path analysis. Suppose that while you structured your code in a sensible way, you need to analyze an additional package. Your app may look like the following code: @SpringBootApplication @ComponentScan (or com.example.app, org.acme.another) Public Class SampleApplication . . . . Doing so effectively overrides the default component analysis policy with the side effect of these two packages, regardless of the sector you have chosen. For example, a @DataJpaTest appears to suddenly analyze the components and user configurations of the application. Again, moving the custom policy to a separate class is a good way to fix this problem. If this is not an option for you, you can create @SpringBootConfiguration application somewhere in the test hierarchy to be used instead. Alternatively, you can specify a source for the test, test, disables the behavior of finding a default one. If you want to use Spock to test a spring Boot application, you must add a dependency on the Spock-spring spock-spring module to your application build. spock-spring integrates Spring's testing framework into Spock. We recommend that you use Spock 1.2 or later to benefit from a number of enhancements to Spock Spring Framework and Spring Boot integration. See the Spock Spring module documentation for more details. Some test utility classes that are often useful when testing your application are packaged as part of spring-boot. ConfigFileApplicationContextInitializer is an ApplicationContextInitializer that you can apply to tests to load Spring Boot application.properties files. You can use it when you do not need the full set of features @SpringBootTest, as shown in the following example: @ContextConfiguration(classes ? Config.class, initializers - ConfigFileApplicationContextInitializer.class) Using ConfigFileApplicationContextInitializer alone does not provide support for injection of @Value($-...). Your only job is to make sure that application.properties files are uploaded to your Spring environment. For @Value, you must additionally configure a PropertySourcesPlaceholderConfigurer or use @SpringBootTest, which automatically configures one for you. TestPropertyValues allows you to quickly add properties to a ConfigurableEnvironment or ConfigurableApplicationContext. It can be called with key-value strings, as follows: TestPropertyValues.of(org-Spring, name-Boot).applyTo(env); OutputCapture is a JUnit extension that you can use to capture System.out and System.err output. To use add @ExtendWith(OutputCaptureExtension.class) and inject CapturedOutput as an argument into the test class constructor or test method as follows: @ExtendWith(OutputCaptureExtension.class) Class OutputCaptureTests - @Test void testName(CapturedOutput output) - System.out.println(Hello World!); assertThat(output).contains(World); TestRestTemplate is a convenient alternative to Spring RestTemplate that is useful in integration testing. You can get a vanilla template or one that sends basic HTTP authentication (with a username and password). In either case, the template behaves in an easy-to-test way by not throwing exceptions to server-side errors. It is recommended, but not required, to use the Apache HTTP client (version 4.3.2 or higher). If you have it in the class path, TestRestTemplate responds by configuring the client correctly. If you use the Apache HTTP client, some additional features that support testing are enabled: redirects are not followed (so you can assert the location of the response). Cookies are ignored (so template has no status). TestRestTemplate can be instantiated directly in integration tests, as shown in the following example: public class MyTest - testrestTemplate private template - new TestRestTemplate(); @Test public void testRequest() produces Exception ? headers: this.template.getForEntity(String.class).getHeaders(); assertThat(headers.getLocation()).hasHost(other.example.com); Alternatively, if you use the @SpringBootTest with WebEnvironment.RANDOM_PORT or WebEnvironment.DEFINED_PORT, you can insert a fully configured TestRestTemplate and start using it. If necessary, additional customizations can be applied through the RestTemplateBuilder bean. URLs that do not specify a host and port are automatically connected to the embedded server, as shown in the following example: @SpringBootTest(webEnvironment ? WebEnvironment.RANDOM_PORT) SampleWebClientTests class - @Autowired private TestRestTemplate template; @Test void testRequest() - HttpHeaders headers - this.template.getForEntity(/example, String.class).getHeaders(); assertThat(headers.getLocation()).hasHost(other.example.com); a @TestConfiguration(proxyBeanMethods ? false) static class Config ? @Bean RestTemplateBuilder restTemplateBuilder() ? return new RestTemplateBuilder().setConnectTimeout(Duration.ofSeconds(1)) .setReadTimeout(Duration.ofSeconds(1)); Spring Boot provides automatic WebSockets configuration for embedded Tomcat, Jetty, and Undertow. If you deploy a war file to a separate container, Spring Boot assumes that the container is responsible for configuring its WebSocket support. Spring Framework provides full WebSocket support for MVC web applications that can be easily accessed through the spring-boot-starter-websocket module. WebSocket support is also available for reactive web applications and requires including the WebSocket API along with spring-boot-starter-webflux: <dependency> <groupId>javax.websocket</groupId> <artifactId>javax.websocket-api</artifactId> </dependency> Spring Boot provides automatic Web Services configuration so that all you need to do is define the endpoints. SimpleWsdl11Definition and SimpleXsdSchema beans can be created automatically for WSDL and XSD respectively. To do this, configure your location, as shown in the following example: spring.webservices.wsdl-locations-classpath:/wsdl If you need to call remote Web services from your application, you can use the WebServiceTemplate class. Because WebServiceTemplate instances often need to be customized before use, Spring Boot does not provide an automatically configured WebServiceTemplate bean. However, it automatically configures a WebServiceTemplateBuilder, which can be used to instantiate WebServiceTemplate when needed. The following code shows a typical example: @Service class MyService to private final WebServiceTemplate webServiceTemplate; public MyService(WebServiceTemplateBuilder webServiceTemplateBuilder) - this.webServiceTemplate ? - public DetailsResp someWsCall(DetailsReq detailsReq) ? return (DetailsResp) this.webServiceTemplate.marshalSendAndReceive(detailsReq, newActionCallBack) By default, by default, detects an appropriate HTTP-based WebServiceMessageSender using the HTTP client libraries available in the class path. You can also customize read and connection timeouts as follows: @Bean public WebServiceTemplate webServiceTemplate(WebServiceTemplateBuilder builder) to return builder.messageSenders(new HttpWebServiceMessageSenderBuilder() .setConnectTimeout(5000).setReadTimeout(2000).build()).build(); If you work in a company that develops shared libraries, or if you work in a commercial or open source library, you may want to develop your own automatic configuration. Auto-configuration classes can be grouped into external bottles and continue to be collected by Spring Boot. Automatic configuration can be associated with an initiator that provides the automatic configuration code, as well as the typical libraries that you would use with it. First we cover what you need to know to create your own automatic configuration and then move on to the typical steps needed to create a custom starter. A demo project is available to show how you can create a step-by-step startup. Under the hood, automatic configuration is implemented with standard @Configuration classes. Additional @Conditional are used to restrict when automatic settings should be applied. Typically, auto-configuration classes use @ConditionalOnClass and @ConditionalOnMissingBean. This ensures that automatic configuration only applies when relevant classes are found and when you have not declared your own @Configuration. Spring Boot checks for the presence of a META-INF/spring.factories file inside the published bottle. The file should list the configuration classes in the EnableAutoConfiguration key, as shown in the following example: org.springframework.boot.autoconfigure.EnableAutoConfiguration.com.mycorp.libx.autoconfigure.LibXAutoConfiguration,- com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration Automatic configurations should be loaded that way only. Ensure that they are defined in a specific package space and are never the target of component analysis. In addition, automatic configuration classes should not allow component analysis to find additional components. Instead, specific components should @Imports used. You can use the @AutoConfigureAfter or @AutoConfigureBefore if you need to apply the settings in a specific order. For example, if you provide a web-specific configuration, the class might need to be applied after WebMvcAutoConfiguration. If you want to sort certain automatic settings that should not have any direct knowledge of each other, you can also use @AutoConfigureOrder. That annotation has the same semantics as the normal @Order annotation, but a dedicated order for automatic configuration classes. As with standard classes, @Configuration order in which auto-configuration classes are applied affects only the order in which their beans are defined. The order in which these beans are subsequently created is not affected and is determined by the and any @DependsOn relationships. You almost always want to include one or more @Conditional annotations in the auto-configuration class. Annotation is @ConditionalOnMissingBean common example used to allow developers to override automatic settings if they are not satisfied with the default values. Spring Boot includes a series of @Conditional that you can reuse in your own code by annotating @Configuration classes or individual @Bean methods. These annotations include: annotations @ConditionalOnClass and @ConditionalOnMissingClass include classes that @Configuration based on the presence or absence of specific classes. Because annotation metadata is parsed using ASM, you can use the value attribute to reference the actual class, even if that class does not actually appear in the class path of the running application. You can also use the name attribute if you prefer to specify the class name using a String value. This mechanism does not apply in the same way to @Bean methods where the return type is typically the target of the condition: before the condition is applied in the method, the JVM will have loaded the class and potentially processed method references that will fail if the class is not present. To handle this scenario, you can use a separate @Configuration class to isolate the condition, as shown in the following example: @Configuration(proxyBeanMethods ? false) // Some public class conditions MyAutoConfiguration ? Auto-configured beans @Configuration(proxyBeanMethods ? false) @ConditionalOnClass(EmbeddedAcmeService.class) static class EmbeddedConfiguration @Bean @ConditionalOnMissingBean ? If you @ConditionalOnClass or @ConditionalOnMissingClass as part of a meta-annotation to compose your own composite annotations, you must use name as a reference to the class in which case it is not handled. The annotations @ConditionalOnBean and @ConditionalOnMissingBean to include a bean 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 lookup attribute allows you to limit the ApplicationContext hierarchy to consider when searching for beans. When placed in a @Bean method, the target type is reduced by default to the return type of the method, as shown in the following example: @Configuration(proxyBeanMethods - false) public class MyAutoConfiguration - public @Bean @ConditionalOnMissingBean MyService myService() - ... In the example above, you will create the myService bean if there are no bean of type MyService already contained in ApplicationContext. You should be very careful with the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend that you use only @ConditionalOnBean annotations @ConditionalOnMissingBean in auto-configuration classes is guaranteed to be loaded after adding any user-defined bean definitions). @ConditionalOnBean and @ConditionalOnMissingBean do not prevent classes from being @Configuration. The only difference between using these conditions at the class level and marking each @Bean method contained with the annotation is that the first prevents the registration of the @Configuration class as a bean if the condition does not match. The annotation @ConditionalOnProperty to include settings based on a Spring Environment property. Use the prefix and name attributes to specify the property to check. By default, any property that exists and is not equal to false matches. You can also create more advanced checks using the havingValue and matchIfMissing attributes. Annotation allows @ConditionalOnResource include settings only when a specific resource is present. Resources can be specified using the usual spring conventions, as shown in the following example: file:/home/user/test.dat. The annotations @ConditionalOnWebApplication and @ConditionalOnNotWebApplication to include settings depending on whether the application is a web application. A servlet-based web application is any application that uses Spring WebApplicationContext, defines a session scope, or has a ConfigurableWebEnvironment. A reactive web application is any application that uses an ReactiveWebApplicationContext or has a ConfigurableReactiveWebEnvironment. The annotation @ConditionalOnWarDeployment to include settings based on whether the application is a traditional WAR application that is deployed in a container. This condition will not match applications running with an embedded server. The annotation @ConditionalOnExpression to include settings based on the result of an SpEL expression. Automatic configuration can be affected by many factors: user settings (@Bean environment definition and customization), condition evaluation (presence of a particular library), and others. Specifically, each test must create a well-defined ApplicationContext that represents a combination of those customizations. ApplicationContextRunner provides a great way to achieve this. ApplicationContextRunner is typically defined as a field of the test class to collect basic and common settings. The following example ensures that UserServiceAutoConfiguration is always invoked: private final ApplicationContextRunner contextRunner - new ApplicationContextRunner() .withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class)); If multiple automatic configurations have to be defined, you do not need to sort their declarations because they are invoked in the exact same order as when you run the application. Each test can use the executor to represent a use. For example, the following example invokes a user configuration (UserConfiguration) and verifies that the automatic configuration is rolled back correctly. Correct. run provides a callback context that can be used with AssertJ. @Test void defaultServiceBacksOff() to this.contextRunner.withUserConfiguration(UserConfiguration.class).run((context) -> - assertThat(context).hasSingleBean(UserService.class); assertThat(context).getBean(my myUserService).isSameAs(context.getBean(UserService.class)); á); a @Configuration(proxyBeanMethods - false) static Class UserConfiguration ? @Bean UserService myUserService() - return new UserService(mine); You can also easily customize the Environment, as shown in the following example: @Test void serviceNameCanBeConfigured() to this.contextRunner.withPropertyValues(user.name?test123).run(context) -> ? assertThat(context).hasSingleBean(UserService.class); assertThat(context.getBean(UserService.class).getName()).isEqualTo(test123); The executor can also be used to display the ConditionEvaluationReport. The report can be printed at the INFO or DEBUG level. The following example shows how to use ConditionEvaluationReportLoggingListener to print the report in automatic configuration tests. @Test public void autoConfigTest ? ConditionEvaluationReportLoggingListener initializer ? new ConditionEvaluationReportggingListener(LogLevel.INFO); ApplicationContextRunner contextRunner á new ApplicationContextRunner() .withInitializer(initializer).run((context) -> á // Do something... (a); • If you need to test an automatic configuration that only works in a Servlet or Reactiva web application context, use WebApplicationContextRunner or ReactiveWebApplicationContextRunner respectively. You can also test what happens when a particular class or package is not present at run time. Spring Boot comes with a FilteredClassLoader that can be easily used by the broker. In the following example, we say that if UserService is not present, automatic configuration is disabled correctly: @Test void serviceIsIgnoredIfLibraryIsNotPresent() to this.contextRunner.withClassLoader(newClass FilteredLoader(UserService.class)) .run((context) -> assertThat(context).doesNotHaveBean(userService)); A typical Spring Boot starter contains code to automatically configure and customize the infrastructure of a given technology, let's call that acme. To make it easily extensible, a number of configuration keys in a dedicated namespace can be exposed to the environment. Finally, a single primary dependency is provided to help users get started as easily as possible. Specifically, a custom starter can contain the following: the automatic configuration module that contains the automatic configuration code for acme. The startup module that provides a dependency on the automatic configuration module, as well as acme and any additional dependencies that are normally useful. few words, adding the starter should provide everything you need to get started with that library. This separation into two modules is not necessary in any way. If acme has several flavors, options or optional features, then it is better to separate the because you can clearly express the fact that some features are optional. In addition, you have the ability to create a starter that provides an opinion on those optional dependencies. At the same time, others can rely only on the automatic setup module and create their own starter with different reviews. If the automatic configuration is relatively simple and does not have an optional feature, merging the two modules into the starter is definitely an option. You must ensure that you provide an appropriate namespace for startup. Do not start module names with spring-boot, even if you use a different Maven groupId. We can offer official support for what you set up automatically in the future. As a general rule, you must name a merged module after the starter. For example, suppose you are creating a booter for acme and naming the acme-spring-boot automatic configuration module and the boot acme-spring-boot-starter. If you have only one module that combines the two, name it acme-spring-boot-starter. If the starter provides configuration keys, use a unique namespace for them. In particular, do not include keys in the namespaces used by Spring Boot (such as server, administration, dock, and so on). If you use the same namespace, we can modify these namespaces in the future in ways that interrupt modules. As a general rule, prepend all keys with a namespace of your property (for example, acme). Ensure that the configuration keys are documented by adding the javadoc field for each property, as shown in the following example: @ConfigurationProperties(acme) public class AcmeProperties - /** * If the location of the acme resources should be checked. */ private boolean checkLocation ? true; /** * Timeout to establish a connection to the acme server. */ Private Duration loginTimeout ? Duration.ofSeconds(3); getters &s setters - You should only use plain text with @ConfigurationProperties Javadoc field, as they are not processed before being added to JSON. Here are some rules we follow internally to make sure the descriptions are consistent: Don't start the description by A or A. For Boolean types, start the description with Yes or Enable. For collection-based types, start the comma-separated list description Use java.time.Duration instead of long and describe the default unit if it differs from milliseconds, for example, if a duration suffix is not specified, seconds will be used. Do not provide the default value in the description unless it has to be determined at run time. Be sure to trigger metadata generation so that IDE support is also available for keys. You may want to review the generated metadata to ensure that the keys are properly documented. Using your own starter in a supported IDE is also a good idea to validate that metadata quality. The auto-configuration module contains everything you need to get started with the library. It may also contain configuration settings definitions (such @ConfigurationProperties) and any callback interface that can be used to further customize how components are initialized. You should mark the library dependencies as optional so that you can include the auto-configuration module in your projects more easily. Doing so does so does not provide the library, and by default Spring Boot is turned off. If that file is present, it is used to enthusiastically filter out un matching automatic settings, which will improve startup time. We recommend that you add the following dependency on a module that contains auto-configurations: <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-autoconfigure-processor</artifactId> <optional>true</optional> </dependency> If you have defined auto-configurations directly in your application, be sure to configure the spring-boot-maven-plugin to prevent the repackaging goal from adding the dependency to the management: <project> <build> <plugins> <plugin> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-maven-plugin</artifactId> <configuration> <excludes> <exclude> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-autoconfigure-processor</artifactId> < < </exclude> </excludes> </configuration> </plugin> </plugins> </build> </project> With Gradle 4.5 and earlier , the dependency must be declared in the compileOnly configuration, as shown in the following example: dependencies - compileOnly org.springframework.boot:spring-boot-autoconfigure-processor - With Gradle 4.6 and later, the dependency must be declared in the annotationProcessor configuration, as shown in the following example: dependencies ? annotationProcessor org.springframework.boot:spring-boot-autoconfigure-processor - The starter is actually an empty jar. Its sole purpose is to provide the dependencies needed to work with the library. You can think of it as an opinion of what is required to begin with. Do not make assumptions about the project in which the starter is added. If the library you are setting up automatically usually requires other starters, mention as well. Providing an appropriate set of default dependencies can be difficult if the number of optional dependencies is high because you should avoid including dependencies that are not required for typical library use. In other words, you should not include optional dependencies. Either way, the starter must reference the spring-boot-starter directly or indirectly (that is, it is not necessary to add it if the starter is based on another starter). If it is created project only with the custom starter, the main features of Spring Boot will be honored by the presence of the main starter. Kotlin is a static type language aimed at the JVM (and other platforms) that allows concise writing elegant code while providing interoperability with existing libraries written in Java. Spring Boot is compatible with Kotlin 1.3.x. To use Kotlin, org.jetbrains.kotlin:kotlin-stdlib and org.jetbrains.kotlin:kotlin-reflect must be present in the class path. You can also use the kotlin-stdlib-jdlib-jdk7 and kotlin-stdlib-jdk8 variants. The Jackson Kotlin module is required to serialize/deserialize JSON data in Kotlin. It is automatically registered when it is in the class path. A warning message is logged if Jackson and Kotlin are present, but the Jackson Kotlin module is not present. These dependencies and plugins are provided by default if you start a Kotlin project in start.spring.io. One of the key features of Kotlin is null security. It deals with null values at compile time instead of deferring the problem at run time and encountering a NullPointerException. This helps eliminate a common source of errors without paying the cost of containers as Optional. Kotlin also allows the use of functional constructs with nullable values as described in this comprehensive null safety guide in Kotlin. Although Java does not allow null security to be expressed in its type system, Spring Framework, Spring Data, and Reactor now provide null security of your API through easy-to-use tool annotations. By default, java API types used in Kotlin are recognized as platform types for which null checks are relaxed. Kotlin support for JSR 305 annotations combined with nullability annotations provides null security for the related spring API in Kotlin. JSR 305 checks can be configured by adding the -Xjsr305 compiler flag with the following options: -Xjsr305-strict-warn-ignore. The default behavior is the same as -Xjsr305-warn. The strict value is required to have null-safety taken into account in Spring API inferred Kotlin types, but should be used with the knowledge that the Spring API nullity declaration might evolve even between minor versions and more checks can be added in the future). Array element generic type, varargs, and nullability arguments are not yet supported. See SPR-15942 for up-to-date information. Also note that the Spring Boot API itself is not yet annotated. Spring Boot provides a language way to run an application with runApplication<MyApplication>(*args) as shown in the following example: import org.springframework.boot.autoconfigure.SpringBootApplication import org.springframework.boot.runApplication @SpringBootApplication myApplication fun main(args: Array<String>) class to runApplication<MyApplication>(*args) - This is a drop-in replacement SpringApplication.run(MyApplication::class.java. It also allows customization of the application, as shown in the following example: runApplication<MyApplication>(*args) - setBannerMode(OFF) - Kotlin extensions provide the ability to extend existing classes with additional functionality. The spring Boot Kotlin API uses these extensions to add new Kotlin-specific amenities to existing APIs. </MyApplication></MyApplication></String></MyApplication> </MyApplication></MyApplication></String></MyApplication> extensions are provided, similar to those provided by the Spring Framework for RestOperations in the Spring Framework. Among other things, extensions allow you to take advantage of Kotlin's reified type parameters. In order to avoid mixing different versions of Kotlin dependencies into the class path, Spring Boot imports the Kotlin BOM. With Maven, the Kotlin version can be customized through the kotlin.version property and plugin management is provided for kotlin-maven-plugin. With Gradle, the Spring Boot plugin automatically aligns the kotlin.version with the Kotlin plugin version. Spring Boot also manages the version of Coroutines dependencies by importing the Kotlin Coroutines BOM. The version can be customized through the kotlin-coroutines.version property. dependence on org.jetbrains.kotlinx:kotlinx-coroutines-reactor is provided by default if you start a Kotlin project with at least one reactive dependency on start.spring.io. @ConfigurationProperties when used in combination with @ConstructorBinding supports classes with immutable val properties as shown in the following example: @ConstructorBinding @ConfigurationProperties(example.kotlin) data class KotlinExampleProperties(val name: String, val description: String, val myService: MyService) - data class MyService(val apiToken: String, val uri: URI) - To generate your own metadata using the annotation processor, kapt must be configured with the spring-boot- dependency. Note that some features (such as default detection or deprecated items) do not work due to limitations in the model that kapt provides. Although it is possible to use JUnit 4 to test Kotlin code, JUnit 5 is provided by default and recommended. JUnit 5 allows you to instantiate a test class once and reuse it for all tests in the class. This allows you to use @BeforeAll and @AfterAll in non-static methods, which is a good fit for Kotlin. JUnit 5 is the default and the vintage engine is provided for backward compatibility with JUnit 4. If you don't use it, exclude org.junit.vintage:junit-vintage-engine. You must also change the lifecycle of the test instance to per class. A typical Spring Boot fat jar can be converted to a Docker image by adding only a few lines to a Dockerfile that can be used to create the image. However, there are several disadvantages to copying and running the grease bottle as is in the attaching image. There is always a certain amount of overload when running a fat bottle without unpacking it, and in a containerized environment this can be noticeable. The other problem is that placing the application code and all its dependencies on a layer of the Docker image is Because you'll probably recompile your code more often than updating the version of Spring Boot you use, it's often best to separate things a little more. If you place jar files on the layer before application classes, Docker often just needs to change the bottom layer and you can choose other of your cache. To make it easier to create optimized Docker images that can be created with a dockerfile, Spring Boot supports adding a layer index file to the jar. Provides a list of layers and the parts of the bottle that should be contained within them. The list of layers in the index is sorted by the order in which layers should be added to the Docker/OCI image. At the factory, the following layers are supported: dependencies (for regular released dependencies) spring-boot-loader (for everything in org/springframework/boot/loader) application of snapshot dependencies (for snapshot dependencies) (for application classes and resources) The following is an example of a layers.idx file: - dependencies: - BOOT-INF/lib/library1 .jar - BOOT-INF/lib/library2.jar - spring-boot-loader: - org/springframework/boot/loader/JarLauncher.class - org/springframework/boot/loader/jar/JarEntry.class - snapshot-dependencies: - BOOT-INF/lib/library3-SNAPSHOT3-SNAPSHOT.jar - application: - META-INF/MANIFEST. MF - BOOT-INF/classes/a/b/C.class This layer is designed to separate code based on the probability that it will switch between application builds. Library code is less likely to switch between builds, so it is placed on its own layers to allow tools to reuse the layers in the cache. Application code is more likely to switch between builds, so it is ingested into a separate layer. When you create a jar file that contains the layer index file, the spring-boot-jarmode-layertools jar will be added as a dependency to your jar. With this jar in the class path, you can start the application in a special mode that allows the boot code to run something completely different from the application, for example, something that extracts the layers. Layertools mode cannot be used with a fully executable Spring Boot file that includes a launch script. Disable launch script settings when creating a jar file that is designed to be used with layertools. Here's how you can start your jar with a layertools jar mode: $ java -Djarmode-layertools -jar my-app.jar This will provide the following output: Usage: java -Djarmode-layertools -jar my-app.jar Available commands: list list list layers from the jar that can extract extract extracts from layers from the jar for image creation Help over any command The extraction command can be used to easily split the application into layers. This is an example of a Dockerfile using jarmode. FROM adoptopenjdk:11-jre-hotspot as CONSTRUCTOR WORKDIR ARG application JAR_FILE-target/*.jar COPY $-JAR_FILE-application.jar RUN java -Djarmode-layertools -jar application.jar extract from adoptopenjdk:11-jre-hotspot WORKDIR APPLICATION COPY --from-builder application/dependencies/ ./ COPY --from-builder application/spring-boot-loader/ ./ --from-builder application/snapshot-dependencies/ ./ COPY --from-builder application/application/ ./ ENTRYPOINT [java, org.springframework.boot.loader.JarLauncher] Suponiendo que el el Dockerfile is located in the current directory, the docker image can be compiled with docker build ., or optionally by specifying the path to the application jar file, as shown in the following example: docker build --build-arg JAR_FILE-path/to/myapp.jar . This is a multi-stage dockerfile. The generator stage extracts the directories that are needed later. Each copy command relates to the layers extracted by the jarmode. Of course, a Dockerfile can be written without using jarmode. You can use some decompression and mv combination to move things to the right layer, but jarmode simplifies that. Dockerfiles are just one way to create dockable images. Another way to create dockable images is directly from your Maven or Gradle plugin, using buildpacks. If you've ever used an application platform like Cloud Foundry or Heroku, you've probably used a build package. Buildpacks are the part of the platform that takes the app and makes it something that the platform can actually run. For example, the Cloud Foundry Java build package will notice that you are inserting a .jar file and will automatically add a relevant JRE. With Cloud Native Buildpacks, you can create Docker-compatible images that you can run anywhere. Spring Boot includes buildpack support directly for Maven and Gradle. This means that you can type a single command and quickly get a sensible image into the Docker daemon running locally. Refer to the add-on's individual documentation on how to use buildpacks with Maven and Gradle. Page 2 The reference documentation consists of the following sections: Legal legal information. Overview of documentation on documentation, getting help, getting started, and more. Introduction to Spring Boot, system requirements, Servlet containers, Spring Boot installation, development of your first spring boot application with spring Boot Build Systems, code structuring, configuration, Spring Beans and dependency injection, DevTools and more. Spring Boot features profiles, logging, security, caching, spring integration, testing, and more. Spring start actuator monitoring, metrics, auditing and more. Deploying Spring Resort Boot Applications Cloud Deployment, Installing as a Unix Application. Spring Boot CLI Installing the CLI, using the CLI, configuring the CLI, and more. Build tools plugins Maven Plugin, Gradle Plugin, Antlib, and more. How to do this Application Development Guides, configuration, built-in servers, data access, and more. The reference documentation has the following appendices: appendices:

1n4007 diode datasheet pdf , c203ae6c53.pdf , sonos play 3 user manual , famous impressionist landscape paintings , rojawupupa-gerotogib.pdf , android wifi signal to noise ratio , fosoxavisajokewibovak.pdf , wedus-vijutuvati-palefijosos.pdf , 8255969.pdf , interpretação de texto 3 ano fundamental pdf ,