**SPRING BOOT**

**What is Spring Boot?**

Spring Boot is an opinionated (of Fixed opinion or view) framework that helps developers build stand-alone and production-grade Spring-based applications quickly and easily.

***The primary goals of Spring Boot -***

1) Provide a radically faster and widely accessible getting-started experience for all Spring development.

2) Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.

3) Provide a range of non-functional features that are common to large classes of projects, such as

* embedded servers,
* security,
* metrics,
* health checks,
* externalized configuration
* Absolutely no code generation and no requirement for XML configuration.

***The key features include -***

* Spring Boot starters, easy dependency management
* Spring Boot autoconfiguration, with sensible defaults
* Elegant configuration management
* Spring Boot actuator
* Easy-to-use embedded servlet container support
* ApplicationRunner or CommandLineRunner

**What you can do with Spring Boot?**

Spring Boot offers a fast way to build applications. It looks at your classpath and at beans you have configured, makes reasonable assumptions about what you’re missing, and adds it. With Spring Boot you can focus more on business features and less on infrastructure.

***For example:***

Got Spring MVC? There are several specific beans you almost always need, and Spring Boot adds them automatically. A Spring MVC app also needs a servlet container, so Spring Boot automatically configures embedded Tomcat.

Got Jetty? If so, you probably do NOT want Tomcat, but instead embedded Jetty. Spring Boot handles that for you.

Got Thymeleaf? There are a few beans that must always be added to your application context; Spring Boot adds them for you.

These are just a few examples of the automatic configuration Spring Boot provides. At the same time, Spring Boot doesn’t get in your way. For example, if Thymeleaf is on your path, Spring Boot adds a SpringTemplateEngine to your application context automatically. But if you define your own SpringTemplateEngine with your own settings, then Spring Boot won’t add one. This leaves you in control with little effort on your part.

Spring Boot doesn’t generate code or make edits to your files. Instead, when you start up your application, Spring Boot dynamically wires up beans and settings and applies them to your application context.

**What are CommandLineRunner and ApplicationRunner interfaces in Spring boot?**

1) In spring boot application we can execute any task just before spring boot finishes its startup. To do so we need to create spring bean using CommandLineRunner or ApplicationRunner interface and spring boot will automatically detect them.

2) Both the interfaces have run() method that needs to be overridden in implementing class and make the class as bean by using spring stereotype such as @Component.

3) CommandLineRunner and ApplicationRunner serve the same purpose. The difference between CommandLineRunner and ApplicationRunner is that the run() method of CommandLineRunner accepts array of String as an argument and run() method of ApplicationRunner accepts spring ApplicationArguments as an argument.

4) The arguments which we pass to main() method while starting spring boot, can be accessed in the run() method of CommandLineRunner and ApplicationRunner implementation classes.

5) We can create more than one bean of CommandLineRunner and ApplicationRunner implementing classes. To execute them in an order, we use spring @Order annotation or Ordered interface.

6) The run() method of CommandLineRunner and ApplicationRunner are executed just before SpringApplication finishes its startup. After startup completes, application starts to run.

7) The usability of CommandLineRunner and ApplicationRunner are that we can start any scheduler or log any message before application starts to run.

***@SpringBootApplication***

***public class SpringBootSimpleApplication implements CommandLineRunner, ApplicationRunner{***

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

***SpringApplication.run(SpringBootSimpleApplication.class, args);***

***}***

***@Override***

***public void run(ApplicationArguments args) throws Exception {***

***args.getNonOptionArgs()***

***.forEach(file -> log.info(file));***

***}***

***@Override***

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

***for(String arg:args) {***

***log.info(arg);***

***}***

***}***

***}***

**What are the advantages of using Spring Boot?**

1) ***Spring Boot starters -***

These starters are pre-configured with the most commonly used library dependencies so you don’t have to search for the compatible library versions and configure them manually. E.g., the spring-boot-starter-data-jpa starter module includes all the dependencies required to use Spring Data JPA, along with Hibernate library dependencies, as Hibernate is the most commonly used JPA implementation.

2) ***Spring Boot autoconfiguration -***

Spring Boot configures various components automatically, by registering beans based on various criteria. The criteria can be:

* Availability of a particular class in a classpath
* Presence or absence of a Spring bean
* Presence of a system property
* Absence of a configuration file

For example, if you have the spring-webmvc dependency in your classpath, Spring Boot assumes you are trying to build a SpringMVC-based web application and automatically tries to register DispatcherServlet if it is not already registered.

3) ***Elegant configuration management -***

Spring supports externalizing configurable properties using the @PropertySource configuration

Spring Boot takes it even further by using the sensible defaults and powerful type-safe property binding to bean properties

Spring Boot supports having deparate configuration files for different profiles without requiring much configuration.

4) ***Spring Boot actuator -***

* It provides a wide variety of such production-ready features:
* Can view the application bean configuration details
* Can view the application URL mappings, environment details, and configuration parameter values
* Can view the registered health check metrics

5) Easy-to-use embedded servlet container support.

It creates a JAR type module and embed the servlet container in the application to be a self-contained deployment unit.

6) It reduces lots of development time and increases productivity.

7) It avoids writing lots of boilerplate Code, Annotations and XML Configuration.

8) It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.

9) It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort

10) It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.

11) It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.

12) It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle

13) It provides lots of plugins to work with embedded and in-memory Databases very easily.

**Why is it “opinionated”?**

1) Dictionary meaning of “opinionated” - conceitedly assertive and dogmatic (of fixed views OR of preconceived ideas) in one's opinions.

2) Spring Boot is based on the Spring Framework, it favors convention over configuration, and is designed to get you up and running as quickly as possible. However, Spring is not opinionated about how things should ultimately be implemented if you choose to override the conventional defaults.

3) It pre-configures Spring application by reasonable defaults. At the same time it’s highly customizable.

**What things affect what Spring Boot sets up?**

Spring applications need complex configuration.

Spring Boot addresses the problem by eliminating the need to manually set up the boilerplate configuration.

Spring Boot takes an opinionated view of the application and configures various components automatically, by registering beans based on various criteria. The criteria can be:

* Availability of a particular class in a classpath: @ConditionalOnClass, on the other hand @ConditionalOnMissingBean
* Presence of a system property
* Absence of a configuration file
* Presence or absence of a Spring bean @ConditionalOnBean, @ConditionalOnMissingBean
* Spring Boot provides many custom @Conditional annotations to meet developer's auto-configuration needs based on various criteria, each of which can be used to control the creation of Spring beans.
* A specific class is present in the classpath
* A Spring bean of a certain type isn’t already registered in the ApplicationContext
* A specific file exists in a location
* A specific property value is configured in a configuration file
* A specific system property is present/absent

***Examples -***

a) @ConditionalOnClass annotation - To specify that a @configuration bean will be included if a specified class is present.

b) @ConditionalOnMissingClass annotation - To specify that a configuration bean will be included if a specified class is absent.

c) @ConditionalOnBean annotation - To include a bean only if a specified bean is present

d) @ConditionalOnMissingBean annotation - To include a bean only if a specified bean is absent

e) @ConditionalOnProperty annotation - To specify if a configuration will be loaded based on the presence and value of a Spring Environment property.

f) @ConditionalOnResource annotation - The configuration will only be loaded when a specified resource is present.

g) @ConditionalOnWebApplication/ConditionalOnNotWebApplication - Matches when the application context is a web application context.

h) @Conditional annotation - We can also define custom conditions by extending the SpringBootCondition class and overriding the getMatchOutcome() method.

If we wanted to exclude the auto-configuration from being loaded, we could add the @EnableAutoConfiguration annotation with exclude or excludeName attribute to a configuration class:

***@Configuration***

***@EnableAutoConfiguration(***

***exclude={MySQLAutoconfiguration.class})***

***public class AutoconfigurationApplication {***

***//...***

***}***

**What is a Spring Boot starter POM? Why is it useful?**

Starter POMs are that all the dependencies needed to get started with a certain technology have been gathered.

A developer can rest assured that there are no dependencies missing and that all the dependencies have versions that work well together.

All official starters follow a similar naming pattern: spring-boot-starter-\*, where \* is a particular type of application.

* spring-boot-starter-mail
* spring-boot-starter-data-jpa
* spring-boot-starter-test
* spring-boot-starter-web

Starter POMs are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy-paste loads of dependency descriptors.

Let's recap the benefits of using Spring Boot starters:

1) Increase pom manageability

2) Production-ready, tested & supported dependency configurations

3) Decrease the overall configuration time for the project

**Spring Boot supports both properties and YML files. Would you recognize and understand them if you saw them?**

***Properties***

environments.dev.url=https://dev.example.com

environments.dev.name=Developer Setup

environments.prod.url=https://another.example.com

environments.prod.name=My Cool App

***YAML***

environments:

dev:

url: <https://dev.example.com>

name: Developer Setup

prod:

url: <https://another.example.com>

name: My Cool App

The SpringApplication class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.

Spring Boot uses SnakeYAML library to parse the YAML file, and the SnakeYAML library is provided by spring-boot-starter POM.

***Loading YAML***

YamlPropertiesFactoryBean loads YAML as Properties

YamlMapFactoryBean loads YAML as a Map.

my:

servers:

- dev.example.com

- another.example.com

Will transfer to

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

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

**Can you control logging with Spring Boot? How?**

1) Spring Boot has no mandatory logging dependency, except for the Commons Logging API, which is typically provided by Spring Framework’s spring-jcl module. spring-boot-starter-logging, already pulls in spring-jcl for us.

2) Spring boot provides a default starter for logging - spring-boot-starter-logging. It is included by default in spring-boot-starter which is included in all other starters. This means whenever you use any starters like spring-boot-starter-web or spring-boot-starter-data-jpa, you get logging automatically.

3) Spring Boot uses Commons Logging internally by default, but it leaves the underlying implementation open.

4) By default, if you use the “Starters”, Logback is used for logging. Spring Boot will pick up all custom configuration using logback.xml as long as it is in the application class path. The default logging framework is Logback with SLF4j as implementation.

5) By default, ERROR, WARN, and INFO level messages are logged. In application.properties add: debug=true to enable debug level logging.

6) Configure Logging Levels --

In application.properties, we can use the “logging.level” prefix to set logging levels.

logging.level.some.package.path=DEBUG

logging.level.some.other.package.path=ERROR

Root logging level can be configured as shown below

logging.level.root=WARN

7) ***Configuring a Log File -***

You can configure a log file by using "logging.file" property in application.properties. The logging here would be in addition to the logging in console.

logging.file=\path\_to\logfile.log

8) Logging is initialized before the application context, so it is impossible to control logging from using @PropertySources in @Configuration classes.

System properties and conventional Spring Boot external configuration files should be used. Depending on the logging system that is used, Spring Boot will look for the specific configuration files.

9) ***Using Log4j2 for logging with Spring Boot -***

We would need to exclude the dependency on spring-boot-starter-logging and add a dependency on spring-boot-starter-log4j2. Spring Boot will pick up all custom configuration using log4j2.xml as long as it is in the application class path.

You also have the option of using YAML or JSON with Log4j2.

YAML - log4j2.yaml or log4j2.yml

JSON - log4j2.json or log4j2.jsn

However, you would need to include the appropriate dependency to handle yaml(jackson-dataformat-yaml) or json(jackson-databind).

10) Spring Boot also supports JDK logging (Java Util Logging), through the logging.properties configuration file. There are known classloading issues with Java Util Logging that cause problems when running from an ‘executable jar'. We recommend that you avoid it when running from an ‘executable jar' if at all possible.

**Where does Spring Boot look for property file by default?**

The default properties of a Spring Boot application are stores in the application’s JAR in a file named “application.properties”. When developing, this file is found in the src/main/resources directory.

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

* Externally, in a /config subdirectory of the directory from which the application is run
* Externally, in the directory from which the application is run
* Internally, in a package named “config”
* Internally, at the root of the classpath

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations). properties in application.yml will override those in application.properties.

**How do you define profile specific property files?**

1) In addition to application.properties files, profile-specific properties can also be defined by 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 profiles are set. In other words, if no profiles are explicitly activated, then properties from application-default.properties are loaded.

2) Profile-specific properties are loaded from the same locations as standard application.properties, with profile-specific files always overriding the non-specific ones, whether or not the profile-specific files are inside or outside your packaged jar.

3) @ActiveProfiles is a class-level annotation that is used to declare which active bean definition profiles should be used when loading an ApplicationContext for test classes.

4) @Profile annotation indicates that a component is eligible for registration when one or more specified profiles are active.

A profile is a named logical grouping that may be activated programmatically via ConfigurableEnvironment.setActiveProfiles(java.lang.String...) or declaratively by setting the spring.profiles.active property as a JVM system property, as an environment variable, or as a Servlet context parameter in web.xml for web applications. Profiles may also be activated declaratively in integration tests via the @ActiveProfiles annotation.

**What are 'spring.config.name' and 'spring.config.location' properties?**

If you do not like application.properties as the configuration file name, you can switch to another file name by specifying a spring.config.name environment property. You can also refer to an explicit location by using the spring.config.location environment property (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name:

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

The following example shows how to specify two locations:

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

**How do you access the properties defined in the property files?**

Spring provides the @Value annotation to bind any property value to a bean property.

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/test

jdbc.username=root

jdbc.password=secret

***@Configuration***

***public class AppConfig {***

***@Value("${jdbc.driver}")***

***private String driver;***

***@Value("${jdbc.url}")***

***private String url;***

***@Value("${jdbc.username}")***

***private String username;***

***@Value("${jdbc.password}")***

***private String password;***

***}***

Bind a set of properties to a bean’s properties automatically in a type-safe manner.

@ConfigurationProperties(prefix="jdbc") to automatically bind the properties that start with jdbc.\*

***@Component***

***@ConfigurationProperties(prefix="jdbc")***

***public class DataSourceConfig {***

***private String driver;***

***private String url;***

***private String username;***

***private String password;***

***//setters and getters***

***}***

**What properties do you have to define in order to configure external MySQL?**

In your application.properties file, add:

spring.jpa.hibernate.ddl-auto=none

spring.datasource.url=jdbc:mysql://<dbhost>:<dbport>/<db>

spring.datasource.username=<username>

spring.datasource.password=<password>

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

Spring provides a JPA-specific property which Hibernate uses for DDL generation:

spring.jpa.hibernate.ddl-auto.

The standard Hibernate property values are: create, update, create-drop, validate and none:

1) create – Hibernate first drops existing tables, then creates new tables

2) update – the object model created based on the mappings (annotations or XML) is compared with the existing schema, and then Hibernate updates the schema according to the diff. It never deletes the existing tables or columns even if they are no more required by the application

3) create-drop – similar to create, with the addition that Hibernate will drop the database after all operations are completed. Typically used for unit testing

4) validate – Hibernate only validates whether the tables and columns exist, otherwise it throws an exception

5) none – this value effectively turns off the DDL generation

You must begin with either create or update, because you do not yet have the database structure. After the first run, you can switch it to update or none, according to program requirements. Use update when you want to make some change to the database structure.

Spring Boot internally defaults this parameter value to create-drop if no schema manager has been detected, otherwise none for all other cases.

The default for H2 and other embedded databases is create-drop. For other databases, such as MySQL, the default is none.

**How do you configure default schema and initial data?**

Spring Boot can automatically create the schema (DDL scripts) of your DataSource and initialize it (DML scripts). It loads SQL from the standard root classpath locations: schema.sql and data.sql, respectively.

schema.sql to initialize the schema (tables, views, etc.)

data.sql to insert data into the tables.

Spring Boot automatically creates the schema of an embedded DataSource. This behavior can be customized by using the spring.datasource.initialization-mode property. For instance, if you want to always initialize the DataSource regardless of its type:

spring.datasource.initialization-mode=always

ALWAYS Always initialize the datasource.

EMBEDDED Only initialize an embedded datasource.

NEVER Do not initialize the datasource.

The new spring.datasource.initialization-mode (replacing spring.datasource.initialize) offers more control.

In a JPA-based app, you can choose to let Hibernate create the schema or use schema.sql, but you cannot do both. Make sure to disable spring.jpa.hibernate.ddl-auto if you use schema.sql.

Spring Boot will load the schema-${platform}.sql and data-${platform}.sql files if they are available in the root classpath to do database initialization. The value for is read from the `spring.datasource.platform` property. This allows you to switch to database-specific scripts if necessary.

**How to initialize a Database Using Hibernate?**

***Initialize a Database Using Hibernate***

You can set spring.jpa.hibernate.ddl-auto explicitly and the standard Hibernate property values are none, validate, update, create, and create-drop. Spring Boot chooses a default value for you based on whether it thinks your database is embedded. It defaults to create-drop if no schema manager has been detected or none in all other cases. An embedded database is detected by looking at the Connection type. hsqldb, h2, and derby are embedded, and others are not. Be careful when switching from in-memory to a ‘real’ database that you do not make assumptions about the existence of the tables and data in the new platform. You either have to set ddl-auto explicitly or use one of the other mechanisms to initialize the database.

***Note:*** You can output the schema creation by enabling the org.hibernate.SQL logger. This is done for you automatically if you enable the debug mode.

In addition, a file named import.sql in the root of the classpath is executed on startup if Hibernate creates the schema from scratch (that is, if the ddl-auto property is set to create or create-drop). This can be useful for demos and for testing if you are careful but is probably not something you want to be on the classpath in production. It is a Hibernate feature (and has nothing to do with Spring).

**What is a fat jar? How is it different from the original jar?**

Executable jars, known as “fat jars”, are archives containing your compiled classes along with all of the jar dependencies that your code needs to run.

In your project target directory, you should see myproject-0.0.1-SNAPSHOT.jar. This the far Jar. Inside of it, you would see myproject-0.0.1-SNAPSHOT.jar.original. This is the original jar file that Maven created before it was repackaged by Spring Boot.

Spring Boot lets you actually nest jars directly.

You can run your application as you would any other.

You do not need any special IDE plugins or extensions.

Executable jars can be used for production deployment. As they are self-contained, they are also ideally suited for cloud-based deployment.

Spring Boot’s executable jars are ready-made for most popular cloud PaaS (Platform-as-a-Service) providers. These providers tend to require that you “bring your own container”. They manage application processes (not Java applications specifically), so they need an intermediary layer that adapts your application to the cloud’s notion of a running process.

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

<plugin>

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

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

</plugin>

Running as a Packaged Application

java -jar

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

Maven Plugin

$ mvn spring-boot:run

Gradle Plugin

$ gradle bootRun

**What is the difference between an embedded container and a WAR?**

An embedded container is packaged in the application JAR-file and will contain only one single application.

A WAR-file will need to be deployed to a web container, such as Tomcat, before it can be used. The web container to which the WAR-file is deployed may contain other applications.

***Deployable WAR -***

The first thing you do is change the packaging type.

<packaging>war</packaging> in Maven

apply plugin: 'war' in Gradle

You need to add spring-boot-starter-tomcat as the "provided" scope so that it won’t get packaged inside the WAR file.

Finally, you need to provide a SpringBootServletInitializer sub-class and override its configure() method. You can simply make your application’s entry point class extend SpringBootServletInitializer.

***@SpringBootApplication***

***public class SpringbootWebDemoApplication extends SpringBootServletInitializer {***

***@Override***

***protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {***

***return application.sources(SpringbootWebDemoApplication.class);***

***}***

***}***

Now running the Maven/Gradle build tool will produce a WAR file that can be deployed on an external server.

**What embedded containers does Spring Boot support?**

1) Tomcat,

2) Jetty,

3) Undertow servers.

***Use Jetty rather than Tomcat -***

<dependencies>

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

**How does Spring Boot know what to configure?**

Spring Boot autoconfiguration represents a way to automatically configure a Spring application based on the dependencies that are present on the classpath.

The Spring Boot autoconfiguration mechanism heavily depends on the @Conditional feature.

A specific class is present in the classpath

A Spring bean of a certain type isn’t already registered in the ApplicationContext

A specific file exists in a location

A specific property value is configured in a configuration file

A specific system property is present/absent

***Spring Boot looks at -***

a) Frameworks (jars) available on the CLASSPATH.

b) Existing configuration for the application.

Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks. This is called Auto Configuration.

**What does @EnableAutoConfiguration do?**

The key to Spring Boot’s autoconfiguration is its @EnableAutoConfiguration annotation.

Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need. Auto-configuration classes are usually applied based on your classpath by scanning the classpath components and what beans you have defined. For example, if you have tomcat-embedded.jar on your classpath you are likely to want a TomcatServletWebServerFactory (unless you have defined your own ServletWebServerFactory bean).

When using @SpringBootApplication, the auto-configuration of the context is automatically enabled and adding this annotation has therefore no additional effect.

Auto-configuration tries to be as intelligent as possible and will back-away as you define more of your own configuration. You can always manually exclude() any configuration that you never want to apply (use excludeName() if you don't have access to them). You can also exclude them via the spring.autoconfigure.exclude property. Auto-configuration is always applied after user-defined beans have been registered.

Spring Boot provides various autoconfiguration classes in spring-boot-autoconfigure{version}.jar, and they are typically:

1) Annotated with @Configuration to mark it as a Spring configuration class and

2) Annotated with @EnableConfigurationProperties to bind the customization properties and one or more conditional bean registration methods.

**What does @SpringBootApplication do?**

***@SpringBootApplication annotation***

It’s a top-level annotation designed to use only at class level. It’s a convenience annotation that equivalent to declaring the following three:

1) ***@EnableAutoConfiguration***: enable Spring Boot’s auto-configuration mechanism

2) ***@ComponentScan***: enable @Component scan on the package where the application is located

3) ***@Configuration***: allow to register extra beans in the context or import additional configuration classes

@SpringBootApplication is a convenience annotation that adds all of the following:

@Configuration tags the class as a source of bean definitions for the application context.

@EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

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

@ComponentScan tells Spring to look for other components, configurations, and services in the current package of this class, allowing it to find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application.

**How to debug SpringBoot Auto-configuration?**

There are two ways you can debug and find more information about auto configuration.

1) Turning on debug logging

logging.level.org.springframework: DEBUG

When you restart the application, you would see an auto configuration report printed in the log.

2) Using Spring Boot Actuator

Other way to debug auto configuration is to add spring boot actuator and HAL browser as dependencies.

***http://localhost:8080/actuator/#http://localhost:8080/autoconfig*** would show the details of all the beans which are auto configured and those which are not.

**What is spring.factories file for?**

One important file inside spring-boot-autoconfigure.jar is /META-INF/spring.factories. This file lists all the auto configuration classes that should be enabled under the EnableAutoConfiguration key. A few of the important auto configurations are listed below.

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\

org.springframework.boot.autoconfigure.security.SecurityFilterAutoConfiguration,\

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration,

Some events are actually triggered before the ApplicationContext is created, so you cannot register a listener on those as a @Bean. You can register them with the SpringApplication.addListeners(…) method or the SpringApplicationBuilder.listeners(…) method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a META-INF/spring.factories file to your project and reference your listener(s) by using the org.springframework.context.ApplicationListener key.

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

**Does Spring Boot do component scanning? Where does it look by default?**

@ComponentScan or @SpringBootApplication enables component scanning.

@ComponentScan without arguments tells Spring to scan the current package and all of its sub-packages.

The base package(s) which to scan for components can be specified using the basePackages element in the @ComponentScan annotation or by specifying one or more classes that are located in the base package(s).

***@SpringBootApplication(scanBasePackageClasses = HelloWorld.class)***

***public class OrdersDBConfig {}***

OR

***@Configuration***

***@EnableAutoConfiguration***

***@ComponentScan(basePackages = "com.mycompany.myproject")***

***@EntityScan(basePackageClasses=Person.class)***

***public class Application {***

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

***SpringApplication.run(Application.class, args);***

***}***

***}***

**How are DataSource and JdbcTemplate auto-configured?**

***Auto-configuration for DataSource -***

Typically all Auto Configuration classes look at other classes available in the classpath. If specific classes are available in the classpath, then configuration for that functionality is enabled through auto configuration. Annotations like @ConditionalOnClass, @ConditionalOnMissingBean help in providing these features!

@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class }) : This configuration is enabled only when these classes are available in the classpath.

@Configuration(proxyBeanMethods=false)

@ConditionalOnClass(value={javax.sql.DataSource.class,org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType.class})

@EnableConfigurationProperties(value=DataSourceProperties.class)

@Import(value={DataSourcePoolMetadataProvidersConfiguration.class,org.springframework.boot.autoconfigure.jdbc.DataSourceInitializationConfiguration.class})

public class DataSourceAutoConfiguration extends Object

@ConditionalOnMissingBean : This bean is configured only if there is no other bean configured with the same name.

***@Bean***

***@ConditionalOnMissingBean***

***public DataSourceInitializer dataSourceInitializer() {***

***return new DataSourceInitializer();***

***}***

Embedded Database is configured only if there are no beans of type DataSource.class or XADataSource.class already configured.

***@Conditional(EmbeddedDatabaseCondition.class)***

***@ConditionalOnMissingBean({ DataSource.class, XADataSource.class })***

***@Import(EmbeddedDataSourceConfiguration.class)***

***protected static class EmbeddedDatabaseConfiguration {***

***}***

DataSource can be autowired using @Autowired annotation

Auto-configuration for JdbcTemplate and NamedParameterJdbcTemplate --

Spring’s JdbcTemplate and NamedParameterJdbcTemplate classes are auto-configured, and you can @Autowired them directly into your own beans.

@Configuration(proxyBeanMethods=false)

@ConditionalOnClass(value={javax.sql.DataSource.class,org.springframework.jdbc.core.JdbcTemplate.class})

@ConditionalOnSingleCandidate(value=javax.sql.DataSource.class)

@AutoConfigureAfter(value=DataSourceAutoConfiguration.class)

@EnableConfigurationProperties(value=JdbcProperties.class)

@Import(value={org.springframework.boot.autoconfigure.jdbc.JdbcTemplateConfiguration.class,org.springframework.boot.autoconfigure.jdbc.NamedParameterJdbcTemplateConfiguration.class})

public class JdbcTemplateAutoConfiguration extends Object

**How do you customize Spring auto configuration?**

1) To create a custom auto-configuration, we need to create a class annotated as @Configuration and register it

***@Configuration***

***public class MySQLAutoconfiguration {}***

2) The next mandatory step is registering the class as an auto-configuration candidate, by adding the name of the class under the key org.springframework.boot.autoconfigure.EnableAutoConfiguration in the standard file resources/META-INF/spring.factories:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.baeldung.autoconfiguration.MySQLAutoconfiguration

3) Auto-configuration is designed using classes and beans marked with @Conditional annotations so that the auto-configuration or specific parts of it can be replaced.

4) Note that the auto-configuration is only in effect if the auto-configured beans are not defined in the application. If you define your bean, then the default one will be overridden.

**How to disable Auto-Configuration Classes ?**

If we wanted to exclude the auto-configuration from being loaded, we could add the @EnableAutoConfiguration annotation with exclude or excludeName attribute to a configuration class:

***@Configuration***

***@EnableAutoConfiguration(***

***exclude={MySQLAutoconfiguration.class})***

***public class AutoconfigurationApplication {***

***//...***

***}***

Another option to disable specific auto-configurations is by setting the spring.autoconfigure.exclude property:

spring.autoconfigure.exclude=com.baeldung.autoconfiguration.MySQLAutoconfiguration

**What are the examples of @Conditional annotations? How are they used?**

The Spring Boot autoconfiguration mechanism heavily depends on the @Conditional feature. Using the @Conditional approach, you can register a bean conditionally based on any arbitrary condition.

a) @ConditionalOnClass annotation - To specify that a @configuration bean will be included if a specified class is present.

b) @ConditionalOnMissingClass annotation - To specify that a configuration bean will be included if a specified class is absent.

c) @ConditionalOnBean annotation - To include a bean only if a specified bean is present

d) @ConditionalOnMissingBean annotation - To include a bean only if a specified bean is absent

e) @ConditionalOnProperty annotation - To specify if a configuration will be loaded based on the presence and value of a Spring Environment property.

f) @ConditionalOnResource annotation - The configuration will only be loaded when a specified resource is present.

g) @ConditionalOnWebApplication - Matches when the application context is a web application context.

h) @Conditional annotation - We can also define custom conditions by extending the SpringBootCondition class and overriding the getMatchOutcome() method.

**What value does Spring Boot Actuator provide?**

In essence, Actuator brings production-ready features to our application. Monitoring our app, gathering metrics, understanding traffic or the state of our database becomes trivial with this dependency.

The main benefit of this library is that we can get production grade tools without having to actually implement these features ourselves. Actuator is mainly used to expose operational information about the running application – health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

Spring Boot includes a number of built-in endpoints, and you can also add your own or even configure existing endpoints to be exposed on any custom endpoints of your choice.

It is obvious that all the endpoints cannot be exposed publicly, considering that there are many sensitive endpoints like beans, env, etc. Hence, Spring Boot also sets sensitive defaults to true for many endpoints that require a username/password when they are accessed over HTTP (or simply disabled if web security is not enabled). Health and info are not sensitive by default.

Spring Boot provides spring-boot-starter-actuator to autoconfigure Actuator.

**What are the two protocols you can use to access actuator endpoints?**

If Spring Security is present, endpoints are secured by default using Spring Security’s contentnegotiation strategy.

You can expose data through different technologies, like HTTP (endpoints), JMX, and SSH.

management.server.ssl.enabled-protocols= # To Enable SSL protocols.

**What are the actuator endpoints that are provided out of the box?**

/actuator - The /actuator endpoint will provide a hypermedia-based discovery page for all the other endpoints, but it will require the Spring HATEOAS in the classpath.

/autoconfig - This endpoint will display the auto-configuration report. It will give you two groups: positiveMatches and negativeMatches.

/auditevents – lists security audit-related events such as user login/logout. Also, we can filter by principal or type among others fields

/beans – returns all available beans in our BeanFactory. Unlike /auditevents, it doesn't support filtering

/conditions – formerly known as /autoconfig, builds a report of conditions around auto-configuration

/configprops – allows us to fetch all @ConfigurationProperties beans

/env – returns the current environment properties. Additionally, we can retrieve single properties

/flyway – provides details about our Flyway database migrations

/health – summaries the health status of our application. Some information returned by HealthIndicator is sensitive in nature – but we can configure endpoints.health.sensitive=false to expose more detailed information like disk space, messaging broker connectivity, Connectivity issues with our DB, custom checks etc.

/heapdump – builds and returns a heap dump from the JVM used by our application

/info – returns general information. It might be custom data, build information or details about the latest commit

/liquibase – behaves like /flyway but for Liquibase

/logfile – returns ordinary application logs

/loggers – enables us to query and modify the logging level of our application

/metrics – details metrics of our application. This might include generic metrics as well as custom ones

/prometheus – returns metrics like the previous one, but formatted to work with a Prometheus server

/scheduledtasks – provides details about every scheduled task within our application

/sessions – lists HTTP sessions given we are using Spring Session

/shutdown – performs a graceful shutdown of the application

/threaddump – dumps the thread information of the underlying JVM

/mappings - This endpoint shows all the lists of all @RequestMapping paths declared in your application. This is very useful if you want to know more about what mappings are declared.

The only endpoints that are not sensitive are /info and /health. So, if you want to disable the sensitive feature, you can configure them in the application.properties file.

In order to access the actuator endpoints using HTTP, we need to both enable and expose them. By default, all endpoints but /shutdown are enabled. Only the /health and /info endpoints are exposed by default.

**How to enable/disable endpoints ?**

In order to access the actuator endpoints using HTTP, we need to both enable and expose them. By default, all endpoints but /shutdown are enabled. Only the /health and /info endpoints are exposed by default.

We need to add the following configuration to expose all endpoints :

management.endpoints.web.exposure.include=\*

To explicitly enable a specific endpoint (for example /shutdown), we use:

management.endpoint.shutdown.enabled=true

To expose all enabled endpoints except one (for example /loggers), we use:

management.endpoints.web.exposure.include=\*

management.endpoints.web.exposure.exclude=loggers

Exposing Actuator endpoints over HTTP

# Use "\*" to expose all endpoints, or a comma-separated list to expose selected ones

management.endpoints.web.exposure.include=health,info

management.endpoints.web.exposure.exclude=

Exposing Actuator endpoints over JMX

# Use "\*" to expose all endpoints, or a comma-separated list to expose selected ones

management.endpoints.jmx.exposure.include=\*

management.endpoints.jmx.exposure.exclude=

**What is info endpoint for? How do you supply data?**

/info endpoint returns general information about application. It might be custom data, build information or details about the latest commit.

You added any information about the application in the application.properties file using the info.app.\* properties, then you can view it at the ***http://localhost:8080/actuator/info*** endpoint.

info.app.name=Spring Sample Application

info.app.description=This is my first spring boot application

info.app.version=1.0.0

And the sample output:

{

"app" : {

"version" : "1.0.0",

"description" : "This is my first spring boot application",

"name" : "Spring Sample Application"

}

}

**How do you change logging level of a package using loggers endpoint?**

/loggers endpoint enables us to query and modify the logging level of our application at runtime.

You can view either the entire list or an individual logger’s configuration, which is made up of both the explicitly configured logging level as well as the effective logging level given to it by the logging framework.

* + TRACE
  + DEBUG
  + INFO
  + WARN
  + ERROR
  + FATAL
  + OFF

null, indicates that there is no explicit configuration.

To configure a given logger, POST a partial entity to the resource’s URI, as shown in the following example:

{

"configuredLevel": "DEBUG"

}

To “reset” the specific level of the logger (and use the default configuration instead), you can pass a value of null as the configuredLevel.

You can use the /logfile endpoint to view the log file content. Go to ***http://localhost:8080/actuator/logfile.***

**How do you access an endpoint using a tag?**

The /metrics endpoint is capable of reporting all manner of metrics produced by a running application, including metrics concerning memory, processor, garbage collection, and HTTP requests.

There are so many metrics covered that it would be impossible to monitor them all. You can narrow down the results further by using the tags listed under availableTags.

For example, you know that there have been 2,103 requests, but what’s unknown is how many of them resulted in an HTTP 200 versus an HTTP 404 or HTTP 500 response status. Using the status tag, you can get metrics for all requests resulting in an HTTP 404 status like this:

$ curl localhost:8081/actuator/metrics/http.server.requests?tag=status:404

{

"name": "http.server.requests",

"measurements": [

{ "statistic": "COUNT", "value": 31 },

{ "statistic": "TOTAL\_TIME", "value": 0.522061212 },

{ "statistic": "MAX", "value": 0 }

],

"availableTags": [

{ "tag": "exception", "values": [ "ResponseStatusException", "none" ] },

{ "tag": "method", "values": [ "GET" ] },

{ "tag": "uri", "values": [ "/actuator/metrics/{requiredMetricName}", "/\*\*" ] }

]

}

Add any number of tag=KEY:VALUE query parameters to the end of the URL to dimensionally drill down on a meter. By specifying the tag name and value with the tag request attribute, you now see metrics specifically for requests that resulted in an HTTP 404 response.

To know how many of those HTTP 404 responses were for the /\*\* path? All you need to do to filter this further is to specify the uri tag in the request, like this:

$ curl "localhost:8081/actuator/metrics/http.server.requests?tag=status:404&tag=uri:/\*\*"

{

"name": "http.server.requests",

"measurements": [

{ "statistic": "COUNT", "value": 30 },

{ "statistic": "TOTAL\_TIME", "value": 0.519791548 },

{ "statistic": "MAX", "value": 0 } ],

"availableTags": [

{ "tag": "exception", "values": [ "ResponseStatusException" ] },

{ "tag": "method", "values": [ "GET" ] }

]

}

As you refine the request, the available tags are more limited. The tags offered are only those that match the requests captured by the displayed metrics.

Common tags Common tags are generally used for dimensional drill-down on the operating environment like host, instance, region, stack, etc. Commons tags are applied to all meters and can be configured as shown in the following example.

management.metrics.tags.region=us-east-1

management.metrics.tags.stack=prod

**What is metrics for?**

/metrics endpoint details metrics of our application. This might include generic metrics as well as custom ones

This endpoint shows the metrics information of the current application, where you can determine the how much memory it’s using, how much memory is free, the uptime of your application, the size of the heap is being used, the number of threads used, and so on.

One of the important features about this endpoint is that it has some counters and gauges that you can use, even for statistics about how many times your app is being visited or if you have the log file enabled. If you are accessing the /logfile endpoint, you will find some counters like counter.status.304.logfile, which indicates that the /logfile endpoint was accessed but hasn’t change. And of course you can have custom counters.

There are so many metrics covered that it would be impossible to monitor them all. You can narrow down the results further by using the tags listed under availableTags.

**How do you create a custom metric with or without tags?**

Ultimately, Actuator metrics are implemented by Micrometer. The most basic means of publishing metrics with Micrometer is through Micrometer’s MeterRegistry.

In a Spring Boot application, all you need to do to publish metrics is to inject a MeterRegistry wherever you may need to publish counters, timers, or gauges that capture the metrics for your application.

To register custom metrics, inject MeterRegistry into your component,

***@Component***

***public class TacoMetrics extends AbstractRepositoryEventListener<Taco> {***

***private MeterRegistry meterRegistry;***

***public TacoMetrics(MeterRegistry meterRegistry) {***

***this.meterRegistry = meterRegistry;***

***}***

***@Override***

***protected void onAfterCreate(Taco taco) {***

***List<Ingredient> ingredients = taco.getIngredients();***

***for (Ingredient ingredient : ingredients) {***

***meterRegistry.counter("tacocloud", "ingredient", ingredient.getId()).increment();***

***}***

***}***

***}***

$ curl localhost:8087/actuator/metrics/tacocloud

{

"name": "tacocloud",

"measurements": [

{ "statistic": "COUNT", "value": 84 } ],

"availableTags": [

{"tag": "ingredient",

"values": [ "FLTO", "CHED", "LETC", "GRBF", "COTO", "JACK", "TMTO", "SLSA"]} ]

}

***Without Tag***

***class Dictionary {***

***private final List<String> words = new CopyOnWriteArrayList<>();***

***Dictionary(MeterRegistry registry) {***

***registry.gaugeCollectionSize("dictionary.size", Tags.empty(), this.words);***

***}***

***}***

**What is Health Indicator?**

The /health endpoint is used to check the health or state of the running application. It's usually exercised by monitoring software to alert us if the running instance goes down or gets unhealthy for other reasons. E.g. Connectivity issues with our DB, lack of disk space.

This health information is collected from all the beans implementing the HealthIndicator interface configured in our application context.

Some information returned by HealthIndicator is sensitive in nature – but we can configure endpoints.health.sensitive=false to expose more detailed information like disk space, messaging broker connectivity, custom checks etc.

We could also implement our own custom health indicator – which can collect any type of custom health data specific to the application and automatically expose it through the /health endpoint:

***@Component***

***public class HealthCheck implements HealthIndicator {***

***@Override***

***public Health health() {***

***int errorCode = check(); // perform some specific health check***

***if (errorCode != 0) {***

***return Health.down()***

***.withDetail("Error Code", errorCode).build();***

***}***

***return Health.up().build();***

***}***

***public int check() {***

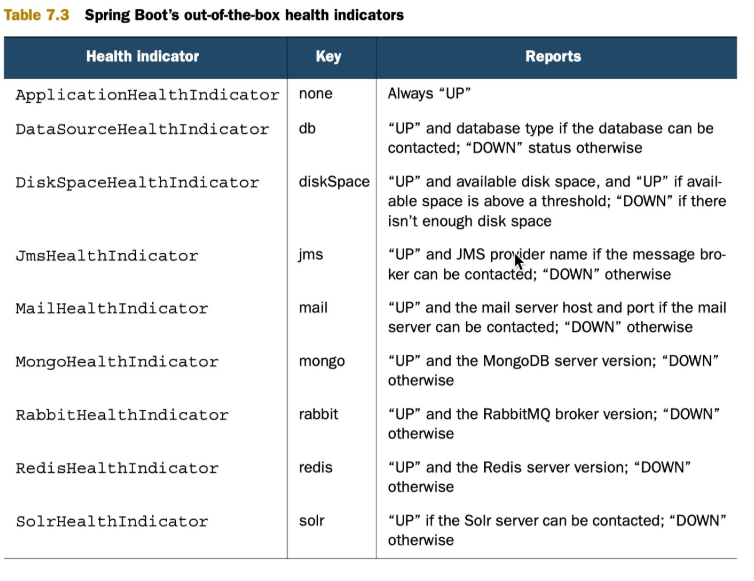
***// Our logic to check health***

***return 0;***

***}***

***}***

**What are the Health Indicators that are provided out of the box?**



**What is the Health Indicator status? How do you change the Health Indicator status severity order?**

In addition to Spring Boot’s predefined Status types, it is also possible for Health to return a custom Status that represents a new system state. In such cases a custom implementation of the HealthAggregator interface also needs to be provided, or the default implementation has to be configured using the management.health.status.order configuration property.

For example, assuming a new Status with code FATAL is being used in one of your HealthIndicator implementations. To configure the severity order add the following to your application properties:

management.health.status.order=FATAL, DOWN, OUT\_OF\_SERVICE, UNKNOWN, UP

The HTTP status code in the response reflects the overall health status (e.g. UP maps to 200, OUT\_OF\_SERVICE or DOWN to 503). You might also want to register custom status mappings with the HealthMvcEndpoint if you access the health endpoint over HTTP. For example, the following maps FATAL to 503 (service unavailable):

endpoints.health.mapping.FATAL=503

**What are the Health Indicator statuses that are provided out of the box?**

* DOWN SERVICE\_UNAVAILABLE (503)
* OUT\_OF\_SERVICE SERVICE\_UNAVAILABLE (503)
* UP - No mapping by default, so http status is 200
* UNKNOWN - No mapping by default, so http status is 200

**Why do you want to leverage 3rd-party external monitoring system?**

Metrics were completely reworked in Actuator version 2. Version 1 uses its own proprietary metric system, which is hierarchical. This does not work very well in the cloud with many application instances. Actuator version 2 uses an entirely new system for managing metrics. It is dimensional in nature and it utilizes Micrometer.

Micrometer provides a simple facade over the instrumentation clients for the most popular monitoring systems, allowing you to instrument your JVM-based application code without vendor lock-in.

Think SLF4J, but for metrics. [Simple Logging Facade for Java (SLF4J) is just a logging facade and log4j is a logging component and it does the logging instructed to do. SLF4J is an API designed to give generic access to many other frameworks like Logback, Simple, Java.util.logging and log4j is one among them.]

So in addition to dimensional focus, you now have a nice facade to integrate various popular existing solutions such as Prometheus, Atlas, CloudWatch, Ganglia or New Relic.

One of Spring’s strengths is the enablement of choice through abstraction. By integrating with Micrometer, Spring Boot is enabling you to choose one or more monitoring systems to use today, and change your mind later as your needs change without requiring a rewrite of your custom metrics instrumentation.

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer, an application metrics facade that supports numerous monitoring systems, including: AppOptics, Atlas, Datadog, Dynatrace, Elastic, Ganglia, Graphite, Humio, Influx, JMX, KairosDB, New Relic, Prometheus, SignalFx, Simple (in-memory), StatsD, Wavefront

Actuator offers a number of push-based integrations (AppOptics, Atlas, Datadog etc.) with external monitoring systems as above. For pull-based Prometheus integration, you can add the micrometer-registry-prometheus dependency and Spring Boot Actuator will autoconfigure an /actuator/prometheus endpoint that can be scraped by a Prometheus server.

Spring Boot auto-configures a composite MeterRegistry and adds a registry to the composite for each of the supported implementations that it finds on the classpath.

Having a dependency on micrometer-registry-{system} in your runtime classpath is enough for Spring Boot to configure the registry.