**Spring Boot Actuator:**

* Spring Boot Actuator is a set of tools provided by the spring team to monitor applications running in production.
* 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.
* **Monitoring our app, gathering metrics, understanding traffic or the state of our database become trivial with this dependency.**

**Spring Boot Actuator Features:**

There are **three** main features of Spring Boot Actuator:

* **Endpoints**
* **Metrics**
* **Audit**

**1) Endpoint:** The actuator endpoints allow us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application chooses **HTTP**, where the Id of the endpoint, along with the prefix of **/actuator,**is mapped to a URL.

For example, the **/health** endpoint provides the basic health information of an application. The actuator, by default, mapped it to **/actuator/health**.

**2) Metrics:** Spring Boot Actuator provides dimensional metrics by integrating with the**micrometer**. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for **timers, gauges, counters, distribution summaries,** and **long task timers** with a dimensional data model.

**3) Audit:** SpringBoot provides a flexible audit framework that publishes events to an **AuditEventRepository.** It automatically publishes the authentication events if spring-security is in execution.

**Enabling Spring Boot Actuator:**

To use the ‘Actuator’ add the following dependency in your application’s project build.

pom.xml

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

**Enabling Endpoints:**

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

**management.endpoint.shutdown.enabled=true**

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

**management.endpoints.enabled-by-default=false**

**management.endpoint.info.enabled=true**

**Exposing Endpoints:**

* By default, only the health endpoint is exposed over HTTP and JMX. Since Endpoints may contain sensitive information, you should carefully consider when to expose them.
* The include property lists the IDs of the endpoints that are exposed. The exclude property lists the IDs of the endpoints that should not be exposed. The exclude property takes precedence over the include property. You can configure both the include and the exclude properties with a list of endpoint IDs.
* For example, to only expose the health and info endpoints over JMX, use the following property:

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

* \* can be used to select all endpoints. For example, to expose everything over HTTP except the env and beans endpoints, use the following properties:

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

**management.endpoints.web.exposure.exclude=env,beans**

**Security:**

* For security purposes, only the /health endpoint is exposed over HTTP by default. You can use the **management.endpoints.web.exposure.include** property to configure the endpoints that are exposed.
* If Spring Security is on the classpath and no other SecurityFilterChain bean is present, all actuators other than /health are secured by Spring Boot auto-configuration. If you define a custom SecurityFilterChain bean, Spring Boot auto-configuration backs off and lets you fully control the actuator access rules.

**Spring Boot Actuator Default Endpoints:**

|  |  |
| --- | --- |
| **Id** | **Usage** |
| Actuator | It provides a hypermedia-based **discovery page** for the other endpoints. It requires Spring HATEOAS to be on the classpath. |
| Autoconfig | It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied. |
| Beans | It is used to display a complete list of all the Spring beans in your application. |
| Configprops | Displays a collated list of all @ConfigurationProperties. |
| Dump | It is used to perform a thread dump. |
| Env | It is used to expose properties from Spring's ConfigurableEnvironment. |
| Flyway | Shows any Flyway database migrations that have been applied. Requires one or more Flyway beans. |
| health | Shows application health information. |
| info | It is used to display arbitrary application info. |
| Loggers | It is used to show and modify the configuration of loggers in the application. |
| liquidbase | It is used to show any Liquibase database migrations that have been applied. |
| Metrics | It is used to show metrics information for the current application. |
| mappings | It is used to display a collated list of all @RequestMapping paths. |
| shutdown | Lets the application be gracefully shutdown. Only works when using jar packaging. Disabled by default. |
| trace | It is used to display trace information. |

For Spring MVC additional endpoints are used,

|  |  |
| --- | --- |
| **Id** | **Usage** |
| Docs | It is used to display documentation, including example requests and responses for the Actuator's endpoints. |
| Headdump | It is used to return a GZip compressed hprof heap dump file. |
| Jolokia | It is used to expose JMX beans over HTTP (when Jolokia is on the classpath). |
| Logfile | It is used to return the contents of the logfile. |

**Custom Endpoints:**

 We can create our own custom actuator endpoints using @Endpoint annotation on a class. Then we have to use @ReadOperation, @WriteOperation, or @DeleteOperation annotations on the methods to expose them as actuator endpoint bean. We can create technology-specific Endpoints using @JmxEndpoint and @WebEndpoint annotations.

**Let’s take an example:**

1. Create custom endpoint with id “appInfo”

@Component  
@Endpoint(id="appInfo")  
public class CustomEndpoint {  
 @Value("${spring.application.maven.version}")  
 private String versionId;  
  
 @Value("${spring.application.name}")  
 private String applicationName;  
  
 @Value("${spring.application.maven.artifactId}")  
 private String artifactId;  
  
 @ReadOperation  
 public Information getApplicationInfo() {  
 Information information = new Information();  
 information.setApplicationName(applicationName);  
 information.setVersionId(versionId);  
 information.setArtifactId(artifactId);  
 information.setStatus("Application status is running");  
 return information;  
  
 }  
}

1. Create Information model class

@Data

public class Information {  
  
 private String versionId;  
 private String artifactId;  
 private String applicationName;  
 private String status;  
}

1. Specify properties to be read in application.properties

spring.application.name=Actuator  
spring.application.maven.version=@project.version@  
spring.application.maven.artifactId=@project.artifactId@

1. Expose our custom endpoint using id ”appInfo”

management.endpoints.web.exposure.include=appInfo

**Customizing endpoint paths:**

* By default base-path of actuator endpoints is /actuator, we can change it to any other value by setting **management.endpoints.web.base-path** in application properties file.

management.server.servlet.context-path=/actuator  
management.endpoints.web.base-path=/endpoints  
management.endpoints.web.exposure.include=metrics,prometheus,info,appInfo

Now, We need to hit endpoint to access health info as: <http://localhost:9090/endpoints/health>

**Health Information:**

* You can use health information to check the status of your running application. It is often used by monitoring software to alert someone when a production system goes down.
* The information exposed by the health endpoint depends on the **management.endpoint.health.show-details** and management.endpoint.health.show-components properties, which can be configured with one of the following values:

|  |  |
| --- | --- |
| Name | Description |
| Never | Details are never shown. |
| when-authorized | Details are shown only to authorized users. Authorized roles can be configured by using management.endpoint.health.roles. |
| Always | Details are shown to all users. |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

* The default value is never. A user is considered to be authorized when they are in one or more of the endpoint’s roles.
* If the endpoint has no configured roles (the default), all authenticated users are considered to be authorized.
* You can configure the roles by using the **management.endpoint.health.roles** property.

**Spring Boot Actuator auto-configured health indicators:**

* When appropriate, Spring Boot auto-configures the HealthIndicators listed in the following table. You can also enable or disable selected indicators by configuring management.health.key.enabled, with the key listed in the following table:

|  |  |  |
| --- | --- | --- |
| Key | Health Indicator | Purpose |
| Cassandra | CassandraDriverHealthIndicator | Verifies Cassandra DB is Up. |
| Diskspace | DiskSpaceHealthIndicator | A [HealthIndicator](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/actuate/health/HealthIndicator.html) that checks available disk space and reports a status of [**Status.DOWN**](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/actuate/health/Status.html#DOWN) when it drops below a configurable threshold. |
| Db | DataSourceHealthIndicator | [HealthIndicator](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/actuate/health/HealthIndicator.html) that tests the status of a [DataSource](https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/javax/sql/DataSource.html) and optionally runs a test query. |
| Jms | JMSHealthIndicator | Checks connection to the JMS Broker. |
| Mail | MailHealthIndicator | Checks that a mail server is up |
| Mongo | MongoHealthIndicator | Checks that a mongo database is up |
| Redis | RedisHealthIndicator | Checks that a Redis Server is up |
| Rabbit | RabbitHealthIndicator | Checks that a Rabbit Server is up |

Additional HealthIndicators are available but are not enabled by default:

|  |  |  |
| --- | --- | --- |
| Key | Name | Description |
| Livenessstate | LivenessStateHealthIndicator | Expose the “liveness” application availability state. |
| Readinessstate | ReadinessStateHealthIndicator | Exposes the “Readiness” application availability state. |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

**Custom Health Indicator:**

Let us define two health Indicators one is up & another is down

**ServiceAHealthIndicator.java**

@Component  
public class ServiceAHealthIndicator implements HealthIndicator {  
  
 private final String message\_key = "Service A";  
  
 @Override  
 public Health health() {  
 if (!isRunningServiceA()) {  
 return Health.*down*().withDetail(message\_key, "Not Available").build();  
 }  
 return Health.*up*().withDetail(message\_key, "Available").build();  
 }  
  
 private Boolean isRunningServiceA() {  
 Boolean isRunning = true;  
 // Logic Skipped  
  
 return isRunning;  
 }  
}

**ServiceBHealthIndicator.java**

@Component  
public class ServiceBHealthIndicator  
 implements HealthIndicator {  
  
 private final String message\_key = "Service B";  
  
 @Override  
 public Health health() {  
 if (!isRunningServiceB()) {  
 return Health.*down*().withDetail(message\_key, "Not Available").build();  
 }  
  
 return Health.*up*().withDetail(message\_key, "Available").build();  
 }  
  
 private Boolean isRunningServiceB() {  
 Boolean isRunning = false;  
 // Logic Skipped  
  
 return isRunning;  
 }  
}

We need to set health details property in application.properties

management.endpoint.health.show-details=always

**Reactive Health Indicator:**

* For reactive applications, such as those that use Spring Web Flux, **ReactiveHealthContributor** provides a non-blocking contract for getting application health.
* Similar to a traditional HealthContributor, health information is collected from the content of a [**ReactiveHealthContributorRegistry**](https://github.com/spring-projects/spring-boot/tree/v3.0.6/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/ReactiveHealthContributorRegistry.java)( by default, all **HealthContributor** and **ReactiveHealthContributor** instances are defined in your ApplicationContext)
* Regular HealthContributors that do not check against a reactive API are executed on the elastic scheduler.
* In a reactive application, you should use the ReactiveHealthContributorRegistry to register and unregister health indicators at runtime. If you need to register a regular HealthContributor, you should wrap it with ReactiveHealthContributor#adapt.

**Custom Reactive Health Indicator:**

@Component  
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {  
  
 @Override  
 public Mono<Health> health() {  
 return doHealthCheck().onErrorResume((exception) ->  
 Mono.just(new Health.Builder().down(exception).build()));  
 }  
  
 private Mono<Health> doHealthCheck() {  
 // perform some specific health check  
 return ...  
 }  
  
}

**Health Groups:**

It is sometimes useful to organize health indicators into groups that you can use for different purposes.

To create a health indicator group, you can use the **management.endpoint.health.group.<name>** property and specify a list of health indicator IDs to include or exclude. For example, to create a group that includes only database indicators you can define the following:

**management.endpoint.health.group.custom.include=db**

**Application Information:**

* Application information exposes various information collected from all [InfoContributor](https://github.com/spring-projects/spring-boot/tree/v3.0.6/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/InfoContributor.java) beans defined in your ApplicationContext.
* Spring Boot includes a number of auto-configured InfoContributor beans, and you can write your own.

**Auto-configured InfoContributors:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Description** | **Prerequsites** |
| Build | BuildInfoContributor | Exposes build information from META-INF/build-info.properties. | A META-INF/build-info.properties resource. |
| Env | EnvironmentInfoContributor | Exposes any property from the **Environment** whose name starts with **info.** | None |
| Git | GitInfoContributor | Exposes git information | A git.properties resource |
| Java | JavaInfoContributor | Exposes Java runtime information. | None |
| Os | OsInfoContributor | Exposes Operating System runtime information. | None |

1. BuildInfoContributor:

To create build-info.properties,

1. Add build goal to spring boot maven plugin

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>build-info</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

1. Now do **maven clean install ,** this will create **target/classes/META-INF/build-info.properties**
2. Hitting <http://localhost:9090/actuator/info> , we can see build info

We can also add custom properties to this build info using the additionalProperties attribute in pom.xml:

<execution>

<goals>

<goal>build-info</goal>

</goals>

<configuration>

<additionalProperties>

<custom.key1>value1</custom.key1>

<custom.key2>value2</custom.key2>

</additionalProperties>

</configuration>

</execution>

If you want to exclude any of the properties that is possible using the excludeInfoProperties configuration.

<configuration>

<excludeInfoProperties>

<infoProperty>artifact</infoProperty>

</excludeInfoProperties>

</configuration>

1. GitInfoContributor:
2. Add the following plugin to pom.xml

<plugin>  
 <groupId>pl.project13.maven</groupId>  
 <artifactId>git-commit-id-plugin</artifactId>  
</plugin>

1. Now do **maven clean install ,** this will create **target/classes/git.properties**
2. Hitting <http://localhost:9090/actuator/info> , we can see Git info

**By default the following information will be exposed:**

* git.branch
* git.commit.id
* git.commit.time

The following management application properties control the Git related information:

| **Application Property** | **Purpose** |
| --- | --- |
| management.info.git.enabled=false | Disables the Git information entirely from the info endpoint |
| management.info.git.mode=full | Displays all the properties from the git.properties file |

To include/exclude specific properties we can add a configuration section like this in pom.xml:

<configuration>

<excludeProperties>

<excludeProperty>time</excludeProperty>

</excludeProperties>

<includeOnlyProperties>

<property>git.commit.id</property>

</includeOnlyProperties>

</configuration>

1. JavaInfoContributor, OsInfoContributor and EnvironmentInfoContributor:

* By default, the env,os and java contributors are disabled.First, we will enable the java contributor & environment contributor by adding the following key-value pair in application.properties:

management.info.java.enabled=true  
management.info.env.enabled=true

management.info.os.enabled=true

* To add environment info

info.app.os=windows10

info.app.version=1

**Spring Boot Actuator Metrics:**

* The Spring Boot Actuator exposes many different monitoring and management endpoints over HTTP and JMX. It includes the all-important metrics capability, by integrating with the [**Micrometer**](https://micrometer.io/) application monitoring framework. Micrometer is a vendor-neutral metrics facade, meaning that metrics can be collected in one common way, but exposed in the format required by many different monitoring systems.
* **The metrics endpoint publishes information about OS and JVM as well as application-level metrics.** Once enabled, we get information such as memory, heap, processors, threads, classes loaded, classes unloaded, and thread pools along with some HTTP metrics as well.
* **To access metrics, we can use endpoint as shown below:**

<http://localhost:9090/actuator/metrics>

**To access particular metric such as process uptime, use endpoint as below:**

<http://localhost:9090/actuator/metrics/process.uptime>

**We can access any metric individually like disk free space, disk total size, process uptime, jvm related metrics etc..**

* Metrics Support various monitoring systems such as data dog, graphite, Prometheus etc..
* For using Prometheus,

1. Add maven dependency:

<dependency>  
 <groupId>io.micrometer</groupId>  
 <artifactId>micrometer-registry-prometheus</artifactId>  
</dependency>

1. Expose endpoint:

management.endpoints.web.exposure.include=metrics,prometheus

1. Use the endpoint: <http://localhost:9090/actuator/prometheus> to check