**Part V. Spring Boot Actuator: Production-ready features**

**52. Enabling Production-ready Features**

The [spring-boot-actuator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator) module provides all of Spring Boot’s production-ready features. The simplest way to enable the features is to add a dependency to the spring-boot-starter-actuator ‘Starter’.

**Definition of Actuator**

An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

To add the actuator to a Maven based project, add the following ‘Starter’ dependency:

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

For Gradle, use the following declaration:

dependencies {

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

}

## 53. Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, thehealth endpoint provides basic application health information.

Each individual endpoint can be [enabled or disabled](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-enabling-endpoints). This controls whether or not the endpoint is created and its bean exists in the application context. To be remotely accessible an endpoint also has to be [exposed via JMX or HTTP](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-exposing-endpoints). Most applications choose HTTP, where the ID of the endpoint along with a prefix of /actuator is mapped to a URL. For example, by default, the health endpoint is mapped to /actuator/health.

The following technology-agnostic endpoints are available:

| **ID** | **Description** | **Enabled by default** |
| --- | --- | --- |
| auditevents | Exposes audit events information for the current application. | Yes |
| beans | Displays a complete list of all the Spring beans in your application. | Yes |
| caches | Exposes available caches. | Yes |
| conditions | Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match. | Yes |
| configprops | Displays a collated list of all @ConfigurationProperties. | Yes |
| env | Exposes properties from Spring’s ConfigurableEnvironment. | Yes |
| flyway | Shows any Flyway database migrations that have been applied. | Yes |
| health | Shows application health information. | Yes |
| httptrace | Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges). | Yes |
| info | Displays arbitrary application info. | Yes |
| integrationgraph | Shows the Spring Integration graph. | Yes |
| loggers | Shows and modifies the configuration of loggers in the application. | Yes |
| liquibase | Shows any Liquibase database migrations that have been applied. | Yes |
| metrics | Shows ‘metrics’ information for the current application. | Yes |
| mappings | Displays a collated list of all @RequestMapping paths. | Yes |
| scheduledtasks | Displays the scheduled tasks in your application. | Yes |
| sessions | Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Not available when using Spring Session’s support for reactive web applications. | Yes |
| shutdown | Lets the application be gracefully shutdown. | No |
| threaddump | Performs a thread dump. | Yes |

If your application is a web application (Spring MVC, Spring WebFlux, or Jersey), you can use the following additional endpoints:

| **ID** | **Description** | **Enabled by default** |
| --- | --- | --- |
| heapdump | Returns an hprof heap dump file. | Yes |
| jolokia | Exposes JMX beans over HTTP (when Jolokia is on the classpath, not available for WebFlux). | Yes |
| logfile | Returns the contents of the logfile (if logging.file or logging.path properties have been set). Supports the use of the HTTP Range header to retrieve part of the log file’s content. | Yes |
| prometheus | Exposes metrics in a format that can be scraped by a Prometheus server. | Yes |

To learn more about the Actuator’s endpoints and their request and response formats, please refer to the separate API documentation ([HTML](https://docs.spring.io/spring-boot/docs/2.1.4.RELEASE/actuator-api/html) or [PDF](https://docs.spring.io/spring-boot/docs/2.1.4.RELEASE/actuator-api/pdf/spring-boot-actuator-web-api.pdf)).

## 53.1 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

|  |
| --- |
| [Note] |
| Disabled endpoints are removed entirely from the application context. If you want to change only the technologies over which an endpoint is exposed, use the [include and exclude properties](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-exposing-endpoints) instead. |

## 53.2 Exposing Endpoints

Since Endpoints may contain sensitive information, careful consideration should be given about when to expose them. The following table shows the default exposure for the built-in endpoints:

| **ID** | **JMX** | **Web** |
| --- | --- | --- |
| auditevents | Yes | No |
| beans | Yes | No |
| caches | Yes | No |
| conditions | Yes | No |
| configprops | Yes | No |
| env | Yes | No |
| flyway | Yes | No |
| health | Yes | Yes |
| heapdump | N/A | No |
| httptrace | Yes | No |
| info | Yes | Yes |
| integrationgraph | Yes | No |
| jolokia | N/A | No |
| logfile | N/A | No |
| loggers | Yes | No |
| liquibase | Yes | No |
| metrics | Yes | No |
| mappings | Yes | No |
| prometheus | N/A | No |
| scheduledtasks | Yes | No |
| sessions | Yes | No |
| shutdown | Yes | No |
| threaddump | Yes | No |

To change which endpoints are exposed, use the following technology-specific include and exclude properties:

| **Property** | **Default** |
| --- | --- |
| management.endpoints.jmx.exposure.exclude |  |
| management.endpoints.jmx.exposure.include | \* |
| management.endpoints.web.exposure.exclude |  |
| management.endpoints.web.exposure.include | info, health |

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 excludeproperty takes precedence over the include property. Both include and exclude properties can be configured with a list of endpoint IDs.

For example, to stop exposing all endpoints over JMX and only expose the health and info endpoints, 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

|  |
| --- |
| [Note] |
| \* has a special meaning in YAML, so be sure to add quotes if you want to include (or exclude) all endpoints, as shown in the following example:  management:  endpoints:  web:  exposure:  include: "\*" |
| [Note] |
| If your application is exposed publicly, we strongly recommend that you also [secure your endpoints](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-security). | |

|  |
| --- |
| [Tip] |
| If you want to implement your own strategy for when endpoints are exposed, you can register an EndpointFilter bean. |

## 53.3 Securing HTTP Endpoints

You should take care to secure HTTP endpoints in the same way that you would any other sensitive URL. If Spring Security is present, endpoints are secured by default using Spring Security’s content-negotiation strategy. If you wish to configure custom security for HTTP endpoints, for example, only allow users with a certain role to access them, Spring Boot provides some convenient RequestMatcher objects that can be used in combination with Spring Security.

A typical Spring Security configuration might look something like the following example:

@Configuration

**public** **class** ActuatorSecurity **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests()

.anyRequest().hasRole("ENDPOINT\_ADMIN")

.and()

.httpBasic();

}

}

The preceding example uses EndpointRequest.toAnyEndpoint() to match a request to any endpoint and then ensures that all have the ENDPOINT\_ADMIN role. Several other matcher methods are also available on EndpointRequest. See the API documentation ([HTML](https://docs.spring.io/spring-boot/docs/2.1.4.RELEASE/actuator-api/html) or [PDF](https://docs.spring.io/spring-boot/docs/2.1.4.RELEASE/actuator-api/pdf/spring-boot-actuator-web-api.pdf)) for details.

If you deploy applications behind a firewall, you may prefer that all your actuator endpoints can be accessed without requiring authentication. You can do so by changing the management.endpoints.web.exposure.include property, as follows:

**application.properties.**

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

Additionally, if Spring Security is present, you would need to add custom security configuration that allows unauthenticated access to the endpoints as shown in the following example:

@Configuration

**public** **class** ActuatorSecurity **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests()

.anyRequest().permitAll();

}

}

## 53.4 Configuring Endpoints

Endpoints automatically cache responses to read operations that do not take any parameters. To configure the amount of time for which an endpoint will cache a response, use its cache.time-to-live property. The following example sets the time-to-live of the beans endpoint’s cache to 10 seconds:

**application.properties.**

management.endpoint.beans.cache.time-to-live=10s

|  |
| --- |
| [Note] |
| The prefix management.endpoint.<name> is used to uniquely identify the endpoint that is being configured. |
| [Note] |
| When making an authenticated HTTP request, the Principal is considered as input to the endpoint and, therefore, the response will not be cached. | |

## 53.5 Hypermedia for Actuator Web Endpoints

A “discovery page” is added with links to all the endpoints. The “discovery page” is available on /actuator by default.

When a custom management context path is configured, the “discovery page” automatically moves from /actuator to the root of the management context. For example, if the management context path is /management, then the discovery page is available from /management. When the management context path is set to /, the discovery page is disabled to prevent the possibility of a clash with other mappings.

## 53.6 CORS Support

[Cross-origin resource sharing](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing) (CORS) is a [W3C specification](https://www.w3.org/TR/cors/) that lets you specify in a flexible way what kind of cross-domain requests are authorized. If you use Spring MVC or Spring WebFlux, Actuator’s web endpoints can be configured to support such scenarios.

CORS support is disabled by default and is only enabled once the management.endpoints.web.cors.allowed-origins property has been set. The following configuration permits GET and POST calls from the example.com domain:

management.endpoints.web.cors.allowed-origins=https://example.com

management.endpoints.web.cors.allowed-methods=GET,POST

|  |
| --- |
| [Tip] |
| See [CorsEndpointProperties](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator-autoconfigure/src/main/java/org/springframework/boot/actuate/autoconfigure/endpoint/web/CorsEndpointProperties.java" \t "_top) for a complete list of options. |

## 53.7 Implementing Custom Endpoints

If you add a @Bean annotated with @Endpoint, any methods annotated with @ReadOperation, @WriteOperation, or @DeleteOperation are automatically exposed over JMX and, in a web application, over HTTP as well. Endpoints can be exposed over HTTP using Jersey, Spring MVC, or Spring WebFlux.

You can also write technology-specific endpoints by using @JmxEndpoint or @WebEndpoint. These endpoints are restricted to their respective technologies. For example, @WebEndpoint is exposed only over HTTP and not over JMX.

You can write technology-specific extensions by using @EndpointWebExtension and @EndpointJmxExtension. These annotations let you provide technology-specific operations to augment an existing endpoint.

Finally, if you need access to web-framework-specific functionality, you can implement Servlet or Spring @Controller and @RestController endpoints at the cost of them not being available over JMX or when using a different web framework.

### 53.7.1 Receiving Input

Operations on an endpoint receive input via their parameters. When exposed via the web, the values for these parameters are taken from the URL’s query parameters and from the JSON request body. When exposed via JMX, the parameters are mapped to the parameters of the MBean’s operations. Parameters are required by default. They can be made optional by annotating them with @org.springframework.lang.Nullable.

Each root property in the JSON request body can be mapped to a parameter of the endpoint. Consider the following JSON request body:

**{**

"name": "test"**,**

"counter": 42

**}**

This can be used to invoke a write operation that takes String name and int counter parameters.

|  |
| --- |
| [Tip] |
| Because endpoints are technology agnostic, only simple types can be specified in the method signature. In particular declaring a single parameter with a custom type defining a name and counter properties is not supported. |
| [Note] |
| To allow the input to be mapped to the operation method’s parameters, Java code implementing an endpoint should be compiled with -parameters, and Kotlin code implementing an endpoint should be compiled with -java-parameters. This will happen automatically if you are using Spring Boot’s Gradle plugin or if you are using Maven and spring-boot-starter-parent. | |

#### Input type conversion

The parameters passed to endpoint operation methods are, if necessary, automatically converted to the required type. Before calling an operation method, the input received via JMX or an HTTP request is converted to the required types using an instance of ApplicationConversionService.

### 53.7.2 Custom Web Endpoints

Operations on an @Endpoint, @WebEndpoint, or @EndpointWebExtension are automatically exposed over HTTP using Jersey, Spring MVC, or Spring WebFlux.

#### Web Endpoint Request Predicates

A request predicate is automatically generated for each operation on a web-exposed endpoint.

#### Path

The path of the predicate is determined by the ID of the endpoint and the base path of web-exposed endpoints. The default base path is /actuator. For example, an endpoint with the ID sessions will use /actuator/sessions as its path in the predicate.

The path can be further customized by annotating one or more parameters of the operation method with @Selector. Such a parameter is added to the path predicate as a path variable. The variable’s value is passed into the operation method when the endpoint operation is invoked.

#### HTTP method

The HTTP method of the predicate is determined by the operation type, as shown in the following table:

| **Operation** | **HTTP method** |
| --- | --- |
| @ReadOperation | GET |
| @WriteOperation | POST |
| @DeleteOperation | DELETE |

#### Consumes

For a @WriteOperation (HTTP POST) that uses the request body, the consumes clause of the predicate is application/vnd.spring-boot.actuator.v2+json, application/json. For all other operations the consumes clause is empty.

#### Produces

The produces clause of the predicate can be determined by the produces attribute of the @DeleteOperation, @ReadOperation, and @WriteOperationannotations. The attribute is optional. If it is not used, the produces clause is determined automatically.

If the operation method returns void or Void the produces clause is empty. If the operation method returns a org.springframework.core.io.Resource, the produces clause is application/octet-stream. For all other operations the produces clause isapplication/vnd.spring-boot.actuator.v2+json, application/json.

#### Web Endpoint Response Status

The default response status for an endpoint operation depends on the operation type (read, write, or delete) and what, if anything, the operation returns.

A @ReadOperation returns a value, the response status will be 200 (OK). If it does not return a value, the response status will be 404 (Not Found).

If a @WriteOperation or @DeleteOperation returns a value, the response status will be 200 (OK). If it does not return a value the response status will be 204 (No Content).

If an operation is invoked without a required parameter, or with a parameter that cannot be converted to the required type, the operation method will not be called and the response status will be 400 (Bad Request).

#### Web Endpoint Range Requests

An HTTP range request can be used to request part of an HTTP resource. When using Spring MVC or Spring Web Flux, operations that return a org.springframework.core.io.Resource automatically support range requests.

|  |
| --- |
| [Note] |
| Range requests are not supported when using Jersey. |

#### Web Endpoint Security

An operation on a web endpoint or a web-specific endpoint extension can receive the current java.security.Principal ororg.springframework.boot.actuate.endpoint.SecurityContext as a method parameter. The former is typically used in conjunction with @Nullable to provide different behaviour for authenticated and unauthenticated users. The latter is typically used to perform authorization checks using its isUserInRole(String) method.

### 53.7.3 Servlet endpoints

A Servlet can be exposed as an endpoint by implementing a class annotated with @ServletEndpoint that also implements Supplier<EndpointServlet>. Servlet endpoints provide deeper integration with the Servlet container but at the expense of portability. They are intended to be used to expose an existing Servlet as an endpoint. For new endpoints, the @Endpoint and @WebEndpoint annotations should be preferred whenever possible.

### 53.7.4 Controller endpoints

@ControllerEndpoint and @RestControllerEndpoint can be used to implement an endpoint that is only exposed by Spring MVC or Spring WebFlux. Methods are mapped using the standard annotations for Spring MVC and Spring WebFlux such as @RequestMapping and @GetMapping, with the endpoint’s ID being used as a prefix for the path. Controller endpoints provide deeper integration with Spring’s web frameworks but at the expense of portability. The @Endpoint and @WebEndpointannotations should be preferred whenever possible.

## 53.8 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 property which can be configured with one of the following values:

| **Name** | **Description** |
| --- | --- |
| never | Details are never shown. |
| when-authorized | Details are only shown to authorized users. Authorized roles can be configured 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. The roles can be configured using the management.endpoint.health.roles property.

|  |
| --- |
| [Note] |
| If you have secured your application and wish to use always, your security configuration must permit access to the health endpoint for both authenticated and unauthenticated users. |

Health information is collected from the content of a [HealthIndicatorRegistry](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthIndicatorRegistry.java" \t "_top) (by default all [HealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthIndicator.java" \t "_top) instances defined in your ApplicationContext. Spring Boot includes a number of auto-configured HealthIndicators and you can also write your own. By default, the final system state is derived by the HealthAggregator which sorts the statuses from each HealthIndicator based on an ordered list of statuses. The first status in the sorted list is used as the overall health status. If no HealthIndicator returns a status that is known to the HealthAggregator, an UNKNOWN status is used.

|  |
| --- |
| [Tip] |
| The HealthIndicatorRegistry can be used to register and unregister health indicators at runtime. |

### 53.8.1 Auto-configured HealthIndicators

The following HealthIndicators are auto-configured by Spring Boot when appropriate:

| **Name** | **Description** |
| --- | --- |
| [CassandraHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/cassandra/CassandraHealthIndicator.java) | Checks that a Cassandra database is up. |
| [CouchbaseHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/couchbase/CouchbaseHealthIndicator.java) | Checks that a Couchbase cluster is up. |
| [DiskSpaceHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/system/DiskSpaceHealthIndicator.java) | Checks for low disk space. |
| [DataSourceHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/jdbc/DataSourceHealthIndicator.java) | Checks that a connection to DataSource can be obtained. |
| [ElasticsearchHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/elasticsearch/ElasticsearchHealthIndicator.java) | Checks that an Elasticsearch cluster is up. |
| [InfluxDbHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/influx/InfluxDbHealthIndicator.java) | Checks that an InfluxDB server is up. |
| [JmsHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/jms/JmsHealthIndicator.java) | Checks that a JMS broker is up. |
| [MailHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/mail/MailHealthIndicator.java) | Checks that a mail server is up. |
| [MongoHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/mongo/MongoHealthIndicator.java) | Checks that a Mongo database is up. |
| [Neo4jHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/neo4j/Neo4jHealthIndicator.java) | Checks that a Neo4j server is up. |
| [RabbitHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/amqp/RabbitHealthIndicator.java) | Checks that a Rabbit server is up. |
| [RedisHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/redis/RedisHealthIndicator.java) | Checks that a Redis server is up. |
| [SolrHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/solr/SolrHealthIndicator.java) | Checks that a Solr server is up. |
| [Tip] | |
| You can disable them all by setting the management.health.defaults.enabled property. |

### 53.8.2 Writing Custom HealthIndicators

To provide custom health information, you can register Spring beans that implement the [HealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthIndicator.java" \t "_top) interface. You need to provide an implementation of the health() method and return a Health response. The Health response should include a status and can optionally include additional details to be displayed. The following code shows a sample HealthIndicator implementation:

**import** org.springframework.boot.actuate.health.Health;

**import** org.springframework.boot.actuate.health.HealthIndicator;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MyHealthIndicator **implements** HealthIndicator {

@Override

**public** Health health() {

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

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

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

}

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

}

}

|  |
| --- |
| [Note] |
| The identifier for a given HealthIndicator is the name of the bean without the HealthIndicator suffix, if it exists. In the preceding example, the health information is available in an entry named my. |

In addition to Spring Boot’s predefined [Status](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/Status.java) types, it is also possible for Health to return a custom Status that represents a new system state. In such cases, a custom implementation of the [HealthAggregator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthAggregator.java" \t "_top) interface also needs to be provided, or the default implementation has to be configured by using the management.health.status.order configuration property.

For example, assume a new Status with code FATAL is being used in one of your HealthIndicator implementations. To configure the severity order, add the following property 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 (for example, UP maps to 200, while OUT\_OF\_SERVICE and DOWN map to 503). You might also want to register custom status mappings if you access the health endpoint over HTTP. For example, the following property maps FATAL to 503 (service unavailable):

management.health.status.http-mapping.FATAL=503

|  |
| --- |
| [Tip] |
| If you need more control, you can define your own HealthStatusHttpMapper bean. |

The following table shows the default status mappings for the built-in statuses:

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

### 53.8.3 Reactive Health Indicators

For reactive applications, such as those using Spring WebFlux, ReactiveHealthIndicator provides a non-blocking contract for getting application health. Similar to a traditional HealthIndicator, health information is collected from the content of a [ReactiveHealthIndicatorRegistry](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/ReactiveHealthIndicatorRegistry.java" \t "_top) (by default all [HealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/HealthIndicator.java" \t "_top) and[ReactiveHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/ReactiveHealthIndicator.java) instances defined in your ApplicationContext. Regular HealthIndicator that do not check against a reactive API are executed on the elastic scheduler.

|  |
| --- |
| [Tip] |
| In a reactive application, The ReactiveHealthIndicatorRegistry can be used to register and unregister health indicators at runtime. |

To provide custom health information from a reactive API, you can register Spring beans that implement the [ReactiveHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/health/ReactiveHealthIndicator.java" \t "_top) interface. The following code shows a sample ReactiveHealthIndicator implementation:

@Component

**public** **class** MyReactiveHealthIndicator **implements** ReactiveHealthIndicator {

@Override

**public** Mono<Health> health() {

**return** doHealthCheck() *//perform some specific health check that returns a Mono<Health>*

.onErrorResume(ex -> Mono.just(**new** Health.Builder().down(ex).build())));

}

}

|  |
| --- |
| [Tip] |
| To handle the error automatically, consider extending from AbstractReactiveHealthIndicator. |

### 53.8.4 Auto-configured ReactiveHealthIndicators

The following ReactiveHealthIndicators are auto-configured by Spring Boot when appropriate:

| **Name** | **Description** |
| --- | --- |
| [CassandraReactiveHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/cassandra/CassandraReactiveHealthIndicator.java) | Checks that a Cassandra database is up. |
| [CouchbaseReactiveHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/couchbase/CouchbaseReactiveHealthIndicator.java) | Checks that a Couchbase cluster is up. |
| [MongoReactiveHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/mongo/MongoReactiveHealthIndicator.java) | Checks that a Mongo database is up. |
| [RedisReactiveHealthIndicator](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/redis/RedisReactiveHealthIndicator.java) | Checks that a Redis server is up. |
| [Tip] | |
| If necessary, reactive indicators replace the regular ones. Also, any HealthIndicator that is not handled explicitly is wrapped automatically. |

## 53.9 Application Information

Application information exposes various information collected from all [InfoContributor](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/InfoContributor.java" \t "_top) beans defined in your ApplicationContext. Spring Boot includes a number of auto-configured InfoContributor beans, and you can write your own.

### 53.9.1 Auto-configured InfoContributors

The following InfoContributor beans are auto-configured by Spring Boot, when appropriate:

| **Name** | **Description** |
| --- | --- |
| [EnvironmentInfoContributor](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/EnvironmentInfoContributor.java) | Exposes any key from the Environment under the info key. |
| [GitInfoContributor](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/GitInfoContributor.java) | Exposes git information if a git.properties file is available. |
| [BuildInfoContributor](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/BuildInfoContributor.java) | Exposes build information if a META-INF/build-info.properties file is available. |
| [Tip] | |
| It is possible to disable them all by setting the management.info.defaults.enabled property. |

### 53.9.2 Custom Application Information

You can customize the data exposed by the info endpoint by setting info.\* Spring properties. All Environment properties under the info key are automatically exposed. For example, you could add the following settings to your application.properties file:

info.app.encoding=UTF-8

info.app.java.source=1.8

info.app.java.target=1.8

|  |
| --- |
| [Tip] |
| Rather than hardcoding those values, you could also [expand info properties at build time](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-properties-and-configuration.html#howto-automatic-expansion).  Assuming you use Maven, you could rewrite the preceding example as follows:  info.app.encoding=@project.build.sourceEncoding@  info.app.java.source=@java.version@  info.app.java.target=@java.version@ |

### 53.9.3 Git Commit Information

Another useful feature of the info endpoint is its ability to publish information about the state of your git source code repository when the project was built. If aGitProperties bean is available, the git.branch, git.commit.id, and git.commit.time properties are exposed.

|  |
| --- |
| [Tip] |
| A GitProperties bean is auto-configured if a git.properties file is available at the root of the classpath. See "[Generate git information](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-build.html#howto-git-info)" for more details. |

If you want to display the full git information (that is, the full content of git.properties), use the management.info.git.mode property, as follows:

management.info.git.mode=full

### 53.9.4 Build Information

If a BuildProperties bean is available, the info endpoint can also publish information about your build. This happens if a META-INF/build-info.properties file is available in the classpath.

|  |
| --- |
| [Tip] |
| The Maven and Gradle plugins can both generate that file. See "[Generate build information](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-build.html#howto-build-info)" for more details. |

### 53.9.5 Writing Custom InfoContributors

To provide custom application information, you can register Spring beans that implement the [InfoContributor](https://github.com/spring-projects/spring-boot/tree/v2.1.4.RELEASE/spring-boot-project/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/info/InfoContributor.java" \t "_top) interface.

The following example contributes an example entry with a single value:

**import** java.util.Collections;

**import** org.springframework.boot.actuate.info.Info;

**import** org.springframework.boot.actuate.info.InfoContributor;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** ExampleInfoContributor **implements** InfoContributor {

@Override

**public** **void** contribute(Info.Builder builder) {

builder.withDetail("example",

Collections.singletonMap("key", "value"));

}

}

If you reach the info endpoint, you should see a response that contains the following additional entry:

**{**

"example": **{**

"key" : "value"

**}**

**}**

**54. Monitoring and Management over HTTP**

If you are developing a web application, Spring Boot Actuator auto-configures all enabled endpoints to be exposed over HTTP. The default convention is to use the id of the endpoint with a prefix of /actuator as the URL path. For example, health is exposed as /actuator/health. TIP: Actuator is supported natively with Spring MVC, Spring WebFlux, and Jersey.

**54.1 Customizing the Management Endpoint Paths**

Sometimes, it is useful to customize the prefix for the management endpoints. For example, your application might already use /actuator for another purpose. You can use the management.endpoints.web.base-path property to change the prefix for your management endpoint, as shown in the following example:

management.endpoints.web.base-path=/manage

The preceding application.properties example changes the endpoint from /actuator/{id} to /manage/{id} (for example, /manage/info).

|  |
| --- |
| [Note] |
| Unless the management port has been configured to [expose endpoints by using a different HTTP port](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-monitoring.html#production-ready-customizing-management-server-port), management.endpoints.web.base-path is relative to server.servlet.context-path. If management.server.port is configured, management.endpoints.web.base-path is relative tomanagement.server.servlet.context-path. |

If you want to map endpoints to a different path, you can use the management.endpoints.web.path-mapping property.

The following example remaps /actuator/health to /healthcheck:

**application.properties.**

management.endpoints.web.base-path=/

management.endpoints.web.path-mapping.health=healthcheck

**54.2 Customizing the Management Server Port**

Exposing management endpoints by using the default HTTP port is a sensible choice for cloud-based deployments. If, however, your application runs inside your own data center, you may prefer to expose endpoints by using a different HTTP port.

You can set the management.server.port property to change the HTTP port, as shown in the following example:

management.server.port=8081

|  |
| --- |
| [Note] |
| On Cloud Foundry, applications only receive requests on port 8080 for both HTTP and TCP routing, by default. If you want to use a custom management port on Cloud Foundry, you will need to explicitly set up the application’s routes to forward traffic to the custom port. |

**54.3 Configuring Management-specific SSL**

When configured to use a custom port, the management server can also be configured with its own SSL by using the various management.server.ssl.\* properties. For example, doing so lets a management server be available over HTTP while the main application uses HTTPS, as shown in the following property settings:

server.port=8443

server.ssl.enabled=true

server.ssl.key-store=classpath:store.jks

server.ssl.key-password=secret

management.server.port=8080

management.server.ssl.enabled=false

Alternatively, both the main server and the management server can use SSL but with different key stores, as follows:

server.port=8443

server.ssl.enabled=true

server.ssl.key-store=classpath:main.jks

server.ssl.key-password=secret

management.server.port=8080

management.server.ssl.enabled=true

management.server.ssl.key-store=classpath:management.jks

management.server.ssl.key-password=secret

**54.4 Customizing the Management Server Address**

You can customize the address that the management endpoints are available on by setting the management.server.address property. Doing so can be useful if you want to listen only on an internal or ops-facing network or to listen only for connections from localhost.

|  |
| --- |
| [Note] |
| You can listen on a different address only when the port differs from the main server port. |

The following example application.properties does not allow remote management connections:

management.server.port=8081

management.server.address=127.0.0.1

**54.5 Disabling HTTP Endpoints**

If you do not want to expose endpoints over HTTP, you can set the management port to -1, as shown in the following example:

management.server.port=-1

This can be achieved using the management.endpoints.web.exposure.exclude property as well, as shown in following example:

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

## 55. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default, Spring Boot exposes management endpoints as JMX MBeans under the org.springframework.boot domain.

## 55.1 Customizing MBean Names

The name of the MBean is usually generated from the id of the endpoint. For example, the health endpoint is exposed as org.springframework.boot:type=Endpoint,name=Health.

If your application contains more than one Spring ApplicationContext, you may find that names clash. To solve this problem, you can set the spring.jmx.unique-names property to true so that MBean names are always unique.

You can also customize the JMX domain under which endpoints are exposed. The following settings show an example of doing so in application.properties:

spring.jmx.unique-names=true

management.endpoints.jmx.domain=com.example.myapp

## 55.2 Disabling JMX Endpoints

If you do not want to expose endpoints over JMX, you can set the management.endpoints.jmx.exposure.exclude property to \*, as shown in the following example:

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

## 55.3 Using Jolokia for JMX over HTTP

Jolokia is a JMX-HTTP bridge that provides an alternative method of accessing JMX beans. To use Jolokia, include a dependency to org.jolokia:jolokia-core. For example, with Maven, you would add the following dependency:

<dependency>

<groupId>org.jolokia</groupId>

<artifactId>jolokia-core</artifactId>

</dependency>

The Jolokia endpoint can then be exposed by adding jolokia or \* to the management.endpoints.web.exposure.include property. You can then access it by using /actuator/jolokia on your management HTTP server.

### 55.3.1 Customizing Jolokia

Jolokia has a number of settings that you would traditionally configure by setting servlet parameters. With Spring Boot, you can use your application.properties file. To do so, prefix the parameter with management.endpoint.jolokia.config., as shown in the following example:

management.endpoint.jolokia.config.debug=true

### 55.3.2 Disabling Jolokia

If you use Jolokia but do not want Spring Boot to configure it, set the management.endpoint.jolokia.enabled property to false, as follows:

management.endpoint.jolokia.enabled=false

## 56. Loggers

Spring Boot Actuator includes the ability to view and configure the log levels of your application at runtime. You can view either the entire list or an individual logger’s configuration, which is made up of both the explicitly configured logging level as well as the effective logging level given to it by the logging framework. These levels can be one of:

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

null indicates that there is no explicit configuration.

## 56.1 Configure a Logger

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. |

## 57. Metrics

Spring Boot Actuator provides dependency management and auto-configuration for [Micrometer](https://micrometer.io/), an application metrics facade that supports numerous monitoring systems, including:

* [AppOptics](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-appoptics)
* [Atlas](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-atlas)
* [Datadog](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-datadog)
* [Dynatrace](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-dynatrace)
* [Elastic](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-elastic)
* [Ganglia](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-ganglia)
* [Graphite](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-graphite)
* [Humio](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-humio)
* [Influx](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-influx)
* [JMX](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-jmx)
* [KairosDB](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-kairos)
* [New Relic](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-newrelic)
* [Prometheus](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-prometheus)
* [SignalFx](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-signalfx)
* [Simple (in-memory)](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-simple)
* [StatsD](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-statsd)
* [Wavefront](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-export-wavefront)

|  |
| --- |
| [Tip] |
| To learn more about Micrometer’s capabilities, please refer to its [reference documentation](https://micrometer.io/docs), in particular the [concepts section](https://micrometer.io/docs/concepts). |

## 57.1 Getting started

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.

Most registries share common features. For instance, you can disable a particular registry even if the Micrometer registry implementation is on the classpath. For instance, to disable Datadog:

management.metrics.export.datadog.enabled=false

Spring Boot will also add any auto-configured registries to the global static composite registry on the Metrics class unless you explicitly tell it not to:

management.metrics.use-global-registry=false

You can register any number of MeterRegistryCustomizer beans to further configure the registry, such as applying common tags, before any meters are registered with the registry:

@Bean

MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {

**return** registry -> registry.config().commonTags("region", "us-east-1");

}

You can apply customizations to particular registry implementations by being more specific about the generic type:

@Bean

MeterRegistryCustomizer<GraphiteMeterRegistry> graphiteMetricsNamingConvention() {

**return** registry -> registry.config().namingConvention(MY\_CUSTOM\_CONVENTION);

}

With that setup in place you can inject MeterRegistry in your components and register metrics:

@Component

**public** **class** SampleBean {

**private** **final** Counter counter;

**public** SampleBean(MeterRegistry registry) {

**this**.counter = registry.counter("received.messages");

}

**public** **void** handleMessage(String message) {

**this**.counter.increment();

*// handle message implementation*

}

}

Spring Boot also [configures built-in instrumentation](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-meter) (i.e. MeterBinder implementations) that you can control via configuration or dedicated annotation markers.

## 57.2 Supported monitoring systems

### 57.2.1 AppOptics

By default, the AppOptics registry pushes metrics to [api.appoptics.com/v1/measurements](https://api.appoptics.com/v1/measurements) periodically. To export metrics to SaaS [AppOptics](https://micrometer.io/docs/registry/appoptics" \t "_top), your API token must be provided:

management.metrics.export.appoptics.api-token=YOUR\_TOKEN

### 57.2.2 Atlas

By default, metrics are exported to [Atlas](https://micrometer.io/docs/registry/atlas) running on your local machine. The location of the [Atlas server](https://github.com/Netflix/atlas) to use can be provided using:

management.metrics.export.atlas.uri=https://atlas.example.com:7101/api/v1/publish

### 57.2.3 Datadog

Datadog registry pushes metrics to [datadoghq](https://www.datadoghq.com/" \t "_top) periodically. To export metrics to [Datadog](https://micrometer.io/docs/registry/datadog" \t "_top), your API key must be provided:

management.metrics.export.datadog.api-key=YOUR\_KEY

You can also change the interval at which metrics are sent to Datadog:

management.metrics.export.datadog.step=30s

### 57.2.4 Dynatrace

Dynatrace registry pushes metrics to the configured URI periodically. To export metrics to [Dynatrace](https://micrometer.io/docs/registry/dynatrace" \t "_top), your API token, device ID, and URI must be provided:

management.metrics.export.dynatrace.api-token=YOUR\_TOKEN

management.metrics.export.dynatrace.device-id=YOUR\_DEVICE\_ID

management.metrics.export.dynatrace.uri=YOUR\_URI

You can also change the interval at which metrics are sent to Dynatrace:

management.metrics.export.dynatrace.step=30s

### 57.2.5 Elastic

By default, metrics are exported to [Elastic](https://micrometer.io/docs/registry/elastic) running on your local machine. The location of the Elastic server to use can be provided using the following property:

management.metrics.export.elastic.host=https://elastic.example.com:8086

### 57.2.6 Ganglia

By default, metrics are exported to [Ganglia](https://micrometer.io/docs/registry/ganglia) running on your local machine. The [Ganglia server](http://ganglia.sourceforge.net/) host and port to use can be provided using:

management.metrics.export.ganglia.host=ganglia.example.com

management.metrics.export.ganglia.port=9649

### 57.2.7 Graphite

By default, metrics are exported to [Graphite](https://micrometer.io/docs/registry/graphite) running on your local machine. The [Graphite server](https://graphiteapp.org/) host and port to use can be provided using:

management.metrics.export.graphite.host=graphite.example.com

management.metrics.export.graphite.port=9004

Micrometer provides a default HierarchicalNameMapper that governs how a dimensional meter id is [mapped to flat hierarchical names](https://micrometer.io/docs/registry/graphite#_hierarchical_name_mapping).

|  |
| --- |
| [Tip] |
| To take control over this behaviour, define your GraphiteMeterRegistry and supply your own HierarchicalNameMapper. An auto-configured GraphiteConfig and Clock beans are provided unless you define your own: |

@Bean

**public** GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig config, Clock clock) {

**return** **new** GraphiteMeterRegistry(config, clock, MY\_HIERARCHICAL\_MAPPER);

}

### 57.2.8 Humio

By default, the Humio registry pushes metrics to [cloud.humio.com](https://cloud.humio.com/) periodically. To export metrics to SaaS [Humio](https://micrometer.io/docs/registry/humio" \t "_top), your API token must be provided:

management.metrics.export.humio.api-token=YOUR\_TOKEN

You should also configure one or more tags to identify the data source to which metrics will be pushed:

management.metrics.export.humio.tags.alpha=a

management.metrics.export.humio.tags.bravo=b

### 57.2.9 Influx

By default, metrics are exported to [Influx](https://micrometer.io/docs/registry/influx) running on your local machine. The location of the [Influx server](https://www.influxdata.com/) to use can be provided using:

management.metrics.export.influx.uri=https://influx.example.com:8086

### 57.2.10 JMX

Micrometer provides a hierarchical mapping to [JMX](https://micrometer.io/docs/registry/jmx), primarily as a cheap and portable way to view metrics locally. By default, metrics are exported to the metrics JMX domain. The domain to use can be provided using:

management.metrics.export.jmx.domain=com.example.app.metrics

Micrometer provides a default HierarchicalNameMapper that governs how a dimensional meter id is [mapped to flat hierarchical names](https://micrometer.io/docs/registry/jmx#_hierarchical_name_mapping).

|  |
| --- |
| [Tip] |
| To take control over this behaviour, define your JmxMeterRegistry and supply your own HierarchicalNameMapper. An auto-configured JmxConfigand Clock beans are provided unless you define your own: |

@Bean

**public** JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock clock) {

**return** **new** JmxMeterRegistry(config, clock, MY\_HIERARCHICAL\_MAPPER);

}

### 57.2.11 KairosDB

By default, metrics are exported to [KairosDB](https://micrometer.io/docs/registry/kairos" \t "_top) running on your local machine. The location of the [KairosDB server](https://kairosdb.github.io/" \t "_top) to use can be provided using:

management.metrics.export.kairos.uri=https://kairosdb.example.com:8080/api/v1/datapoints

### 57.2.12 New Relic

New Relic registry pushes metrics to [New Relic](https://micrometer.io/docs/registry/new-relic) periodically. To export metrics to [New Relic](https://newrelic.com/), your API key and account id must be provided:

management.metrics.export.newrelic.api-key=YOUR\_KEY

management.metrics.export.newrelic.account-id=YOUR\_ACCOUNT\_ID

You can also change the interval at which metrics are sent to New Relic:

management.metrics.export.newrelic.step=30s

### 57.2.13 Prometheus

[Prometheus](https://micrometer.io/docs/registry/prometheus) expects to scrape or poll individual app instances for metrics. Spring Boot provides an actuator endpoint available at /actuator/prometheus to present a [Prometheus scrape](https://prometheus.io/) with the appropriate format.

|  |
| --- |
| [Tip] |
| The endpoint is not available by default and must be exposed, see [exposing endpoints](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-exposing-endpoints) for more details. |

Here is an example scrape\_config to add to prometheus.yml:

scrape\_configs:

- job\_name: 'spring'

metrics\_path: '/actuator/prometheus'

static\_configs:

- targets: ['HOST:PORT'**]**

### 57.2.14 SignalFx

SignalFx registry pushes metrics to [SignalFx](https://micrometer.io/docs/registry/signalfx" \t "_top) periodically. To export metrics to [SignalFx](https://signalfx.com/" \t "_top), your access token must be provided:

management.metrics.export.signalfx.access-token=YOUR\_ACCESS\_TOKEN

You can also change the interval at which metrics are sent to SignalFx:

management.metrics.export.signalfx.step=30s

### 57.2.15 Simple

Micrometer ships with a simple, in-memory backend that is automatically used as a fallback if no other registry is configured. This allows you to see what metrics are collected in the [metrics endpoint](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#production-ready-metrics-endpoint).

The in-memory backend disables itself as soon as you’re using any of the other available backend. You can also disable it explicitly:

management.metrics.export.simple.enabled=false

### 57.2.16 StatsD

The StatsD registry pushes metrics over UDP to a StatsD agent eagerly. By default, metrics are exported to a [StatsD](https://micrometer.io/docs/registry/statsd" \t "_top) agent running on your local machine. The StatsD agent host and port to use can be provided using:

management.metrics.export.statsd.host=statsd.example.com

management.metrics.export.statsd.port=9125

You can also change the StatsD line protocol to use (default to Datadog):

management.metrics.export.statsd.flavor=etsy

### 57.2.17 Wavefront

Wavefront registry pushes metrics to [Wavefront](https://micrometer.io/docs/registry/wavefront" \t "_top) periodically. If you are exporting metrics to [Wavefront](https://www.wavefront.com/" \t "_top) directly, your API token must be provided:

management.metrics.export.wavefront.api-token=YOUR\_API\_TOKEN

Alternatively, you may use a Wavefront sidecar or an internal proxy set up in your environment that forwards metrics data to the Wavefront API host:

management.metrics.export.wavefront.uri=proxy://localhost:2878

|  |
| --- |
| [Tip] |
| If publishing metrics to a Wavefront proxy (as described in [the documentation](https://docs.wavefront.com/proxies_installing.html)), the host must be in the proxy://HOST:PORT format. |

You can also change the interval at which metrics are sent to Wavefront:

management.metrics.export.wavefront.step=30s

## 57.3 Supported Metrics

Spring Boot registers the following core metrics when applicable:

* JVM metrics, report utilization of:
  + Various memory and buffer pools
  + Statistics related to garbage collection
  + Threads utilization
  + Number of classes loaded/unloaded
* CPU metrics
* File descriptor metrics
* Kafka consumer metrics
* Log4j2 metrics: record the number of events logged to Log4j2 at each level
* Logback metrics: record the number of events logged to Logback at each level
* Uptime metrics: report a gauge for uptime and a fixed gauge representing the application’s absolute start time
* Tomcat metrics
* [Spring Integration](https://docs.spring.io/spring-integration/docs/current/reference/html/system-management-chapter.html#micrometer-integration) metrics

### 57.3.1 Spring MVC Metrics

Auto-configuration enables the instrumentation of requests handled by Spring MVC. When management.metrics.web.server.auto-time-requests is true, this instrumentation occurs for all requests. Alternatively, when set to false, you can enable instrumentation by adding @Timed to a request-handling method:

@RestController

@Timed 1

**public** **class** MyController {

@GetMapping("/api/people")

@Timed(extraTags = { "region", "us-east-1" }) 2

@Timed(value = "all.people", longTask = true) 3

**public** List<Person> listPeople() { ... }

}

|  |  |
| --- | --- |
| [1](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#CO1-1) | A controller class to enable timings on every request handler in the controller. |
| [2](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#CO1-2) | A method to enable for an individual endpoint. This is not necessary if you have it on the class, but can be used to further customize the timer for this particular endpoint. |
| [3](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#CO1-3) | A method with longTask = true to enable a long task timer for the method. Long task timers require a separate metric name, and can be stacked with a short task timer. |

By default, metrics are generated with the name, http.server.requests. The name can be customized by setting the management.metrics.web.server.requests-metric-name property.

By default, Spring MVC-related metrics are tagged with the following information:

| **Tag** | **Description** |
| --- | --- |
| exception | Simple class name of any exception that was thrown while handling the request. |
| method | Request’s method (for example, GET or POST) |
| outcome | Request’s outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT\_ERROR, and 5xx is SERVER\_ERROR |
| status | Response’s HTTP status code (for example, 200 or 500) |
| uri | Request’s URI template prior to variable substitution, if possible (for example, /api/person/{id}) |

To customize the tags, provide a @Bean that implements WebMvcTagsProvider.

### 57.3.2 Spring WebFlux Metrics

Auto-configuration enables the instrumentation of all requests handled by WebFlux controllers and functional handlers.

By default, metrics are generated with the name http.server.requests. You can customize the name by setting the management.metrics.web.server.requests-metric-name property.

By default, WebFlux-related metrics are tagged with the following information:

| **Tag** | **Description** |
| --- | --- |
| exception | Simple class name of any exception that was thrown while handling the request. |
| method | Request’s method (for example, GET or POST) |
| outcome | Request’s outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT\_ERROR, and 5xx is SERVER\_ERROR |
| status | Response’s HTTP status code (for example, 200 or 500) |
| uri | Request’s URI template prior to variable substitution, if possible (for example, /api/person/{id}) |

To customize the tags, provide a @Bean that implements WebFluxTagsProvider.

### 57.3.3 Jersey Server Metrics

Auto-configuration enables the instrumentation of requests handled by the Jersey JAX-RS implementation. When management.metrics.web.server.auto-time-requests is true, this instrumentation occurs for all requests. Alternatively, when set to false, you can enable instrumentation by adding @Timed to a request-handling method:

@Component

@Path("/api/people")

@Timed 1

**public** **class** Endpoint {

@GET

@Timed(extraTags = { "region", "us-east-1" }) 2

@Timed(value = "all.people", longTask = true) 3

**public** List<Person> listPeople() { ... }

}

|  |  |
| --- | --- |
| [1](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#CO2-1) | On a resource class to enable timings on every request handler in the resource. |
| [2](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#CO2-2) | On a method to enable for an individual endpoint. This is not necessary if you have it on the class, but can be used to further customize the timer for this particular endpoint. |
| [3](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html#CO2-3) | On a method with longTask = true to enable a long task timer for the method. Long task timers require a separate metric name, and can be stacked with a short task timer. |

By default, metrics are generated with the name, http.server.requests. The name can be customized by setting the management.metrics.web.server.requests-metric-name property.

By default, Jersey server metrics are tagged with the following information:

| **Tag** | **Description** |
| --- | --- |
| exception | Simple class name of any exception that was thrown while handling the request. |
| method | Request’s method (for example, GET or POST) |
| outcome | Request’s outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT\_ERROR, and 5xx is SERVER\_ERROR |
| status | Response’s HTTP status code (for example, 200 or 500) |
| uri | Request’s URI template prior to variable substitution, if possible (for example, /api/person/{id}) |

To customize the tags, provide a @Bean that implements JerseyTagsProvider.

### 57.3.4 HTTP Client Metrics

Spring Boot Actuator manages the instrumentation of both RestTemplate and WebClient. For that, you have to get injected with an auto-configured builder and use it to create instances:

* RestTemplateBuilder for RestTemplate
* WebClient.Builder for WebClient

It is also possible to apply manually the customizers responsible for this instrumentation, namely MetricsRestTemplateCustomizer and MetricsWebClientCustomizer.

By default, metrics are generated with the name, http.client.requests. The name can be customized by setting the management.metrics.web.client.requests-metric-name property.

By default, metrics generated by an instrumented client are tagged with the following information:

* method, the request’s method (for example, GET or POST).
* uri, the request’s URI template prior to variable substitution, if possible (for example, /api/person/{id}).
* status, the response’s HTTP status code (for example, 200 or 500).
* clientName, the host portion of the URI.

To customize the tags, and depending on your choice of client, you can provide a @Bean that implements RestTemplateExchangeTagsProvider orWebClientExchangeTagsProvider. There are convenience static functions in RestTemplateExchangeTags and WebClientExchangeTags.

### 57.3.5 Cache Metrics

Auto-configuration enables the instrumentation of all available Caches on startup with metrics prefixed with cache. Cache instrumentation is standardized for a basic set of metrics. Additional, cache-specific metrics are also available.

The following cache libraries are supported:

* Caffeine
* EhCache 2
* Hazelcast
* Any compliant JCache (JSR-107) implementation

Metrics are tagged by the name of the cache and by the name of the CacheManager that is derived from the bean name.

|  |
| --- |
| [Note] |
| Only caches that are available on startup are bound to the registry. For caches created on-the-fly or programmatically after the startup phase, an explicit registration is required. A CacheMetricsRegistrar bean is made available to make that process easier. |

### 57.3.6 DataSource Metrics

Auto-configuration enables the instrumentation of all available DataSource objects with a metric named jdbc. Data source instrumentation results in gauges representing the currently active, maximum allowed, and minimum allowed connections in the pool. Each of these gauges has a name that is prefixed by jdbc.

Metrics are also tagged by the name of the DataSource computed based on the bean name.

|  |
| --- |
| [Tip] |
| By default, Spring Boot provides metadata for all supported data sources; you can add additional DataSourcePoolMetadataProvider beans if your favorite data source isn’t supported out of the box. See DataSourcePoolMetadataProvidersConfiguration for examples. |

Also, Hikari-specific metrics are exposed with a hikaricp prefix. Each metric is tagged by the name of the Pool (can be controlled with spring.datasource.name).

### 57.3.7 Hibernate Metrics

Auto-configuration enables the instrumentation of all available Hibernate EntityManagerFactory instances that have statistics enabled with a metric namedhibernate.

Metrics are also tagged by the name of the EntityManagerFactory that is derived from the bean name.

To enable statistics, the standard JPA property hibernate.generate\_statistics must be set to true. You can enable that on the auto-configured EntityManagerFactory as shown in the following example:

spring.jpa.properties.hibernate.generate\_statistics=true

### 57.3.8 RabbitMQ Metrics

Auto-configuration will enable the instrumentation of all available RabbitMQ connection factories with a metric named rabbitmq.

## 57.4 Registering custom metrics

To register custom metrics, inject MeterRegistry into your component, as shown in the following example:

**class** Dictionary {

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

Dictionary(MeterRegistry registry) {

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

}

*// …*

}

If you find that you repeatedly instrument a suite of metrics across components or applications, you may encapsulate this suite in a MeterBinder implementation. By default, metrics from all MeterBinder beans will be automatically bound to the Spring-managed MeterRegistry.

## 57.5 Customizing individual metrics

If you need to apply customizations to specific Meter instances you can use the io.micrometer.core.instrument.config.MeterFilter interface. By default, allMeterFilter beans will be automatically applied to the micrometer MeterRegistry.Config.

For example, if you want to rename the mytag.region tag to mytag.area for all meter IDs beginning with com.example, you can do the following:

@Bean

**public** MeterFilter renameRegionTagMeterFilter() {

**return** MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");

}

### 57.5.1 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

The example above adds region and stack tags to all meters with a value of us-east-1 and prod respectively.

|  |
| --- |
| [Note] |
| The order of common tags is important if you are using Graphite. As the order of common tags cannot be guaranteed using this approach, Graphite users are advised to define a custom MeterFilter instead. |

### 57.5.2 Per-meter properties

In addition to MeterFilter beans, it’s also possible to apply a limited set of customization on a per-meter basis using properties. Per-meter customizations apply to any all meter IDs that start with the given name. For example, the following will disable any meters that have an ID starting with example.remote

management.metrics.enable.example.remote=false

The following properties allow per-meter customization:

**Table 57.1. Per-meter customizations**

| **Property** | **Description** |
| --- | --- |
| management.metrics.enable | Whether to deny meters from emitting any metrics. |
| management.metrics.distribution.percentiles-histogram | Whether to publish a histogram suitable for computing aggregable (across dimension) percentile approximations. |
| management.metrics.distribution.minimum-expected-value,management.metrics.distribution.maximum-expected-value | Publish less histogram buckets by clamping the range of expected values. |
| management.metrics.distribution.percentiles | Publish percentile values computed in your application |
| management.metrics.distribution.sla | Publish a cumulative histogram with buckets defined by your SLAs. |

For more details on concepts behind percentiles-histogram, percentiles and sla refer to the ["Histograms and percentiles" section](https://micrometer.io/docs/concepts#_histograms_and_percentiles) of the micrometer documentation.

## 57.6 Metrics endpoint

Spring Boot provides a metrics endpoint that can be used diagnostically to examine the metrics collected by an application. The endpoint is not available by default and must be exposed, see [exposing endpoints](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-exposing-endpoints) for more details.

Navigating to /actuator/metrics displays a list of available meter names. You can drill down to view information about a particular meter by providing its name as a selector, e.g. /actuator/metrics/jvm.memory.max.

|  |
| --- |
| [Tip] |
| The name you use here should match the name used in the code, not the name after it has been naming-convention normalized for a monitoring system it is shipped to. In other words, if jvm.memory.max appears as jvm\_memory\_max in Prometheus because of its snake case naming convention, you should still use jvm.memory.max as the selector when inspecting the meter in the metrics endpoint. |

You can also add any number of tag=KEY:VALUE query parameters to the end of the URL to dimensionally drill down on a meter, e.g./actuator/metrics/jvm.memory.max?tag=area:nonheap.

|  |
| --- |
|  |
| The reported measurements are the sum of the statistics of all meters matching the meter name and any tags that have been applied. So in the example above, the returned "Value" statistic is the sum of the maximum memory footprints of "Code Cache", "Compressed Class Space", and "Metaspace" areas of the heap. If you just wanted to see the maximum size for the "Metaspace", you could add an additional tag=id:Metaspace, i.e./actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace. |

## 58. Auditing

Once Spring Security is in play, Spring Boot Actuator has a flexible audit framework that publishes events (by default, “authentication success”, “failure” and “access denied” exceptions). This feature can be very useful for reporting and for implementing a lock-out policy based on authentication failures. To customize published security events, you can provide your own implementations of AbstractAuthenticationAuditListener and AbstractAuthorizationAuditListener.

You can also use the audit services for your own business events. To do so, either inject the existing AuditEventRepository into your own components and use that directly or publish an AuditApplicationEvent with the Spring ApplicationEventPublisher (by implementing ApplicationEventPublisherAware).

## 59. HTTP Tracing

Tracing is automatically enabled for all HTTP requests. You can view the httptrace endpoint and obtain basic information about the last 100 request-response exchanges.

## 59.1 Custom HTTP tracing

To customize the items that are included in each trace, use the management.trace.http.include configuration property. For advanced customization, consider registering your own HttpExchangeTracer implementation.

By default, an InMemoryHttpTraceRepository that stores traces for the last 100 request-response exchanges is used. If you need to expand the capacity, you can define your own instance of the InMemoryHttpTraceRepository bean. You can also create your own alternative HttpTraceRepository implementation.

**60. Process Monitoring**

In the spring-boot module, you can find two classes to create files that are often useful for process monitoring:

* ApplicationPidFileWriter creates a file containing the application PID (by default, in the application directory with a file name of application.pid).
* WebServerPortFileWriter creates a file (or files) containing the ports of the running web server (by default, in the application directory with a file name ofapplication.port).

By default, these writers are not activated, but you can enable:

* [By Extending Configuration](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-process-monitoring.html#production-ready-process-monitoring-configuration)
* [Section 60.2, “Programmatically”](https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-process-monitoring.html#production-ready-process-monitoring-programmatically)

**60.1 Extending Configuration**

In the META-INF/spring.factories file, you can activate the listener(s) that writes a PID file, as shown in the following example:

org.springframework.context.ApplicationListener=\

org.springframework.boot.context.ApplicationPidFileWriter,\

org.springframework.boot.web.context.WebServerPortFileWriter

**60.2 Programmatically**

You can also activate a listener by invoking the SpringApplication.addListeners(…​) method and passing the appropriate Writer object. This method also lets you customize the file name and path in the Writer constructor.

## 61. Cloud Foundry Support

Spring Boot’s actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The /cloudfoundryapplicationpath provides an alternative secured route to all @Endpoint beans.

The extended support lets Cloud Foundry management UIs (such as the web application that you can use to view deployed applications) be augmented with Spring Boot actuator information. For example, an application status page may include full health information instead of the typical “running” or “stopped” status.

|  |
| --- |
| [Note] |
| The /cloudfoundryapplication path is not directly accessible to regular users. In order to use the endpoint, a valid UAA token must be passed with the request. |

## 61.1 Disabling Extended Cloud Foundry Actuator Support

If you want to fully disable the /cloudfoundryapplication endpoints, you can add the following setting to your application.properties file:

**application.properties.**

management.cloudfoundry.enabled=false

## 61.2 Cloud Foundry Self-signed Certificates

By default, the security verification for /cloudfoundryapplication endpoints makes SSL calls to various Cloud Foundry services. If your Cloud Foundry UAA or Cloud Controller services use self-signed certificates, you need to set the following property:

**application.properties.**

management.cloudfoundry.skip-ssl-validation=true

## 61.3 Custom context path

If the server’s context-path has been configured to anything other than /, the Cloud Foundry endpoints will not be available at the root of the application. For example, ifserver.servlet.context-path=/app, Cloud Foundry endpoints will be available at /app/cloudfoundryapplication/\*.

If you expect the Cloud Foundry endpoints to always be available at /cloudfoundryapplication/\*, regardless of the server’s context-path, you will need to explicitly configure that in your application. The configuration will differ depending on the web server in use. For Tomcat, the following configuration can be added:

@Bean

**public** TomcatServletWebServerFactory servletWebServerFactory() {

**return** **new** TomcatServletWebServerFactory() {

@Override

**protected** **void** prepareContext(Host host,

ServletContextInitializer[] initializers) {

**super**.prepareContext(host, initializers);

StandardContext child = **new** StandardContext();

child.addLifecycleListener(**new** Tomcat.FixContextListener());

child.setPath("/cloudfoundryapplication");

ServletContainerInitializer initializer = getServletContextInitializer(

getContextPath());

child.addServletContainerInitializer(initializer, Collections.emptySet());

child.setCrossContext(true);

host.addChild(child);

}

};

}

**private** ServletContainerInitializer getServletContextInitializer(String contextPath) {

**return** (c, context) -> {

Servlet servlet = **new** GenericServlet() {

@Override

**public** **void** service(ServletRequest req, ServletResponse res)

**throws** ServletException, IOException {

ServletContext context = req.getServletContext()

.getContext(contextPath);

context.getRequestDispatcher("/cloudfoundryapplication").forward(req,

res);

}

};

context.addServlet("cloudfoundry", servlet).addMapping("/\*");

};

}