spring by Pivotal

# Spring Boot Actuator: características listas para producción

Volver al índice

Spring Boot incluye una serie de características adicionales para ayudarlo a monitorear y administrar su aplicación cuando la lleva a producción. Puede elegir administrar y monitorear su aplicación utilizando puntos finales HTTP o con JMX. La auditoría, el estado y la recopilación de métricas también se pueden aplicar automáticamente a su aplicación.

# 1. Habilitar características listas para producción

El `spring-boot-actuator` módulo proporciona todas las funciones listas para producción de Spring Boot. La forma más sencilla de habilitar las funciones es agregar una dependencia al `spring-boot-starter-actuator` 'Starter'.

### Definición de actuador

Un actuador es un término de fabricación que se refiere a un dispositivo mecánico para mover o controlar algo. Los actuadores pueden generar una gran cantidad de movimiento a partir de un pequeño cambio.

Para agregar el actuador a un proyecto basado en Maven, agregue la siguiente dependencia 'Starter':

XML

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

Para Gradle, use la siguiente declaración:

GROOVY

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

# 2. Puntos finales

Los puntos finales del actuador le permiten monitorear e interactuar con su aplicación. Spring Boot incluye varios puntos finales integrados y le permite agregar los suyos propios. Por ejemplo, el `health` punto final proporciona información básica sobre el estado de la aplicación.

Cada punto final individual se puede habilitar o deshabilitar . Esto controla si se crea o no el punto final y si su bean existe en el contexto de la aplicación. Para ser accesible de forma remota, un punto final también debe estar expuesto a través de JMX o HTTP . La mayoría de las aplicaciones eligen HTTP, donde la ID del punto final junto con un prefijo de `/actuator` se asigna a una URL. Por ejemplo, de forma predeterminada, el `health` punto final se asigna a `/actuator/health` .

Están disponibles los siguientes puntos finales independientes de la tecnología:

| CARNÉ DE IDENTIDAD | Descripción | Habilitado por defecto |
|---|---|---|
| `auditevents` | Expone información de eventos de auditoría para la aplicación actual. | Si. Requiere un `AuditEventRepository` frijol |
| `beans` | Muestra una lista completa de todos los beans de Spring en su aplicación. | si |
| `caches` | Expone cachés disponibles. | si |
| `conditions` | Muestra las condiciones que se evaluaron en las clases de configuración y autoconfiguración y las razones por las que coincidieron o no. | si |
| `configprops` | Muestra una lista clasificada de todos `@ConfigurationProperties` . | si |
| `env` | Expone propiedades de Spring's `ConfigurableEnvironment` . | si |
| `flyway` | Muestra las migraciones de la base de datos Flyway que se han aplicado. | si |
| `health` | Muestra información de salud de la aplicación. | si |

| CARNÉ DE IDENTIDAD | Descripción | Habilitado por defecto |
|---|---|---|
| `httptrace` | Muestra información de rastreo HTTP (de manera predeterminada, los últimos 100 intercambios de solicitud-respuesta HTTP). | Si. Requiere un `HttpTraceRepository` frijol |
| `info` | Muestra información arbitraria de la aplicación. | si |
| `integrationgraph` | Muestra el gráfico de Spring Integration. | si |
| `loggers` | Muestra y modifica la configuración de los registradores en la aplicación. | si |
| `liquibase` | Muestra las migraciones de la base de datos de Liquibase que se han aplicado. | si |
| `metrics` | Muestra información de 'métricas' para la aplicación actual. | si |
| `mappings` | Muestra una lista clasificada de todas las `@RequestMapping` rutas. | si |
| `scheduledtasks` | Muestra las tareas programadas en su aplicación. | si |
| `sessions` | Permite la recuperación y eliminación de sesiones de usuario de un almacén de sesiones respaldado por Spring Session. No está disponible cuando se utiliza el soporte de Spring Session para aplicaciones web reactivas. | si |
| `shutdown` | Permite que la aplicación se cierre correctamente. | No |
| `threaddump` | Realiza un volcado de subprocesos. | si |

Si su aplicación es una aplicación web (Spring MVC, Spring WebFlux o Jersey), puede usar los siguientes puntos finales adicionales:

| CARNÉ DE IDENTIDAD | Descripción | Habilitado por defecto |
|---|---|---|
| | | |

| CARNÉ DE IDENTIDAD | Descripción | Habilitado por defecto |
|---|---|---|
| `heapdump` | Devuelve un `hprof` archivo de volcado de montón. | si |
| `jolokia` | Expone los beans JMX sobre HTTP (cuando Jolokia está en el classpath, no disponible para WebFlux). | si |
| `logfile` | Devuelve el contenido del archivo de registro (si `logging.file.name` o `logging.file.path` propiedades se han establecido). Admite el uso del `Range` encabezado HTTP para recuperar parte del contenido del archivo de registro. | si |
| `prometheus` | Expone métricas en un formato que puede ser raspado por un servidor Prometheus. | si |

Para obtener más información sobre los puntos finales del Actuador y sus formatos de solicitud y respuesta, consulte la documentación API separada ( HTML o PDF ).

## 2.1. Habilitación de puntos finales

De forma predeterminada, todos los puntos finales excepto `shutdown` están habilitados. Para configurar la habilitación de un punto final, use su `management.endpoint.<id>.enabled` propiedad. El siguiente ejemplo habilita el `shutdown` punto final:

PROPERTIES

```
management.endpoint.shutdown.enabled=true
```

Si prefiere que la habilitación de punto final sea opcional en lugar de excluirse, establezca la `management.endpoints.enabled-by-default` propiedad `false` y use las `enabled` propiedades de punto final individuales para volver a optar. El siguiente ejemplo habilita el `info` punto final y deshabilita todos los demás puntos finales:

PROPERTIES

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

Los puntos finales deshabilitados se eliminan por completo del contexto de la aplicación. Si desea cambiar solo las tecnologías sobre las que se expone un punto final, utilice las propiedades `include` y `exclude` .

## 2.2. Puntos finales de exposición

Dado que los puntos finales pueden contener información confidencial, se debe considerar cuidadosamente cuándo exponerlos. La siguiente tabla muestra la exposición predeterminada para los puntos finales integrados:

| CARNÉ DE IDENTIDAD | JMX | Web |
|---|---|---|
| `auditevents` | si | No |
| `beans` | si | No |
| `caches` | si | No |
| `conditions` | si | No |
| `configprops` | si | No |
| `env` | si | No |
| `flyway` | si | No |
| `health` | si | si |
| `heapdump` | N / A | No |
| `httptrace` | si | No |
| `info` | si | si |
| `integrationgraph` | si | No |
| `jolokia` | N / A | No |
| `logfile` | N / A | No |
| `loggers` | si | No |
| `liquibase` | si | No |

| CARNÉ DE IDENTIDAD | JMX | Web |
|---|---|---|
| `metrics` | si | No |
| `mappings` | si | No |
| `prometheus` | N / A | No |
| `scheduledtasks` | si | No |
| `sessions` | si | No |
| `shutdown` | si | No |
| `threaddump` | si | No |

Para cambiar qué puntos finales están expuestos, use las siguientes propiedades `include` y `exclude` propiedades específicas de la tecnología :

| Propiedad | Defecto |
|---|---|
| `management.endpoints.jmx.exposure.exclude` | |
| `management.endpoints.jmx.exposure.include` | `*` |
| `management.endpoints.web.exposure.exclude` | |
| `management.endpoints.web.exposure.include` | `info, health` |

La `include` propiedad enumera los ID de los puntos finales que están expuestos. La `exclude` propiedad enumera los ID de los puntos finales que no deben exponerse. La `exclude` propiedad tiene prioridad sobre la `include` propiedad. Ambos `include` y las `exclude` propiedades se pueden configurar con una lista de ID de punto final.

Por ejemplo, para dejar de exponer todos los puntos finales sobre JMX y solo exponer los puntos finales `health` y `info` , use la siguiente propiedad:

PROPERTIES

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

`*` se puede usar para seleccionar todos los puntos finales. Por ejemplo, para exponer todo a través de HTTP excepto los puntos finales `env` y `beans` , use las siguientes propiedades:

PROPERTIES

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

`*` tiene un significado especial en YAML, así que asegúrese de agregar comillas si desea incluir (o excluir) todos los puntos finales, como se muestra en el siguiente ejemplo:

YAML

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Si su aplicación se expone públicamente, le recomendamos encarecidamente que también asegure sus puntos finales .

Si desea implementar su propia estrategia para cuando los puntos finales están expuestos, puede registrar un `EndpointFilter` bean.

## 2.3. Asegurar puntos finales HTTP

Debe asegurarse de proteger los puntos finales HTTP de la misma manera que lo haría con cualquier otra URL sensible. Si Spring Security está presente, los puntos finales están protegidos de forma predeterminada mediante la estrategia de negociación de contenido de Spring Security. Si desea configurar la seguridad personalizada para los puntos finales HTTP, por ejemplo, solo permita que los usuarios con un determinado rol accedan a ellos, Spring Boot proporciona algunos `RequestMatcher` objetos convenientes que se pueden usar en combinación con Spring Security.

Una configuración típica de Spring Security podría parecerse al siguiente ejemplo:

JAVA

```java
@Configuration(proxyBeanMethods = false)
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests) ->
                requests.anyRequest().hasRole("ENDPOINT_ADMIN"));
        http.httpBasic();
    }

}
```

El ejemplo anterior usa `EndpointRequest.toAnyEndpoint()` para hacer coincidir una solicitud con cualquier punto final y luego asegura que todos tengan el `ENDPOINT_ADMIN` rol. Varios otros métodos de comparación también están disponibles en `EndpointRequest` . Consulte la documentación de la API ( HTML o PDF ) para más detalles.

Si implementa aplicaciones detrás de un firewall, puede preferir que se pueda acceder a todos los puntos finales de su actuador sin requerir autenticación. Puede hacerlo cambiando la `management.endpoints.web.exposure.include` propiedad, de la siguiente manera:

**application.properties**

PROPERTIES

```properties
management.endpoints.web.exposure.include=*
```

Además, si Spring Security está presente, necesitará agregar una configuración de seguridad personalizada que permita el acceso no autenticado a los puntos finales como se muestra en el siguiente ejemplo:

JAVA

```java
@Configuration(proxyBeanMethods = false)
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests((requests) ->
            requests.anyRequest().permitAll());
    }

}
```

## 2.4. Configurar puntos finales

Los puntos finales almacenan automáticamente en caché las respuestas a las operaciones de lectura que no toman ningún parámetro. Para configurar la cantidad de tiempo durante el cual un punto final almacenará en caché una respuesta, use su `cache.time-to-live` propiedad. El siguiente ejemplo establece el tiempo de vida de la `beans` memoria caché del punto final en 10 segundos:

**application.properties**

PROPERTIES

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

El prefijo `management.endpoint.<name>` se usa para identificar de forma exclusiva el punto final que se está configurando.

Al realizar una solicitud HTTP autenticada, `Principal` se considera como entrada al punto final y, por lo tanto, la respuesta no se almacenará en caché.

## 2.5. Hipermedia para puntos finales web de actuadores

Se agrega una "página de descubrimiento" con enlaces a todos los puntos finales. La "página de descubrimiento" está disponible `/actuator` de forma predeterminada.

Cuando se configura una ruta de contexto de administración personalizada, la "página de descubrimiento" se mueve automáticamente desde `/actuator` la raíz del contexto de administración. Por ejemplo, si la ruta del contexto de administración es `/management` , entonces la página de descubrimiento está disponible desde `/management` . Cuando la ruta del contexto de administración se establece en `/` , la página de descubrimiento se deshabilita para evitar la posibilidad de un conflicto con otras asignaciones.

## 2.6. Soporte CORS

El intercambio de recursos de origen cruzado (CORS) es una especificación W3C que le permite especificar de manera flexible qué tipo de solicitudes de dominio cruzado están

autorizadas. Si usa Spring MVC o Spring WebFlux, los puntos finales web de Actuator se pueden configurar para admitir tales escenarios.

El soporte de CORS está deshabilitado de manera predeterminada y solo se habilita una vez que `management.endpoints.web.cors.allowed-origins` se ha establecido la propiedad. La siguiente configuración permite `GET` y `POST` llama desde el `example.com` dominio:

PROPERTIES

```
management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

Consulte CorsEndpointProperties para obtener una lista completa de opciones.

## 2.7. Implementación de puntos finales personalizados

Si agrega un `@Bean` anotado con `@Endpoint` , cualquier método anotado con `@ReadOperation` , `@WriteOperation` o `@DeleteOperation` se expone automáticamente a través de JMX y, en una aplicación web, también a través de HTTP. Los puntos finales pueden exponerse a través de HTTP utilizando Jersey, Spring MVC o Spring WebFlux.

También puede escribir puntos finales específicos de la tecnología utilizando `@JmxEndpoint` o `@WebEndpoint` . Estos puntos finales están restringidos a sus respectivas tecnologías. Por ejemplo, `@WebEndpoint` se expone solo a través de HTTP y no a través de JMX.

Puede escribir extensiones específicas de la tecnología utilizando `@EndpointWebExtension` y `@EndpointJmxExtension` . Estas anotaciones le permiten proporcionar operaciones específicas de la tecnología para aumentar un punto final existente.

Finalmente, si necesita acceso a la funcionalidad específica del marco web, puede implementar Servlet o Spring `@Controller` y `@RestController` puntos finales a costa de que no estén disponibles en JMX o cuando use un marco web diferente.

### 2.7.1. Recibiendo entrada

Las operaciones en un punto final reciben información a través de sus parámetros. Cuando se exponen a través de la web, los valores de estos parámetros se toman de los parámetros de consulta de la URL y del cuerpo de solicitud JSON. Cuando se exponen a través de JMX, los parámetros se asignan a los parámetros de las operaciones de MBean. Los parámetros son

obligatorios por defecto. Se pueden hacer opcionales anotándolos con `@org.springframework.lang.Nullable`.

Cada propiedad raíz en el cuerpo de la solicitud JSON se puede asignar a un parámetro del punto final. Considere el siguiente cuerpo de solicitud JSON:

```JSON
{
    "name": "test",
    "counter": 42
}
```

Esto se puede usar para invocar una operación de escritura que toma `String name` y `int counter` parámetros.

> Como los puntos finales son independientes de la tecnología, solo se pueden especificar tipos simples en la firma del método. En particular la que se declara un solo parámetro con un tipo personalizado que define una `name` y `counter` propiedades no es compatible.

> Para permitir que la entrada se asigne a los parámetros del método de operación, se debe compilar el código Java que implementa un punto final, y se debe compilar el `-parameters` código Kotlin que implementa un punto final `-java-parameters`. Esto sucederá automáticamente si está utilizando el complemento Gradle de Spring Boot o si está utilizando Maven y `spring-boot-starter-parent`.

### Conversión de tipo de entrada

Los parámetros pasados a los métodos de operación de punto final se convierten, si es necesario, automáticamente al tipo requerido. Antes de llamar a un método de operación, la entrada recibida a través de JMX o una solicitud HTTP se convierte a los tipos requeridos utilizando una instancia `ApplicationConversionService`, así como cualquiera `Converter` o `GenericConverter` beans calificados `@EndpointConverter`.

## 2.7.2. Puntos finales web personalizados

Las operaciones en un `@Endpoint` , `@WebEndpoint` o `@EndpointWebExtension` se exponen automáticamente a través de HTTP utilizando Jersey, Spring MVC o Spring WebFlux.

## Predicados de solicitud de punto final web

Se genera automáticamente un predicado de solicitud para cada operación en un punto final expuesto a la web.

## Camino

La ruta del predicado está determinada por la ID del punto final y la ruta base de los puntos finales expuestos a la web. La ruta base predeterminada es `/actuator` . Por ejemplo, un punto final con la ID `sessions` se usará `/actuator/sessions` como su ruta en el predicado.

La ruta se puede personalizar aún más anotando uno o más parámetros del método de operación con `@Selector` . Tal parámetro se agrega al predicado de ruta como una variable de ruta. El valor de la variable se pasa al método de operación cuando se invoca la operación de punto final. Si desea capturar todos los elementos de ruta restantes, puede agregar `@Selector(Match=ALL_REMAINING)` al último parámetro y convertirlo en un tipo que sea compatible con la conversión a `String[]` .

## Método HTTP

El método HTTP del predicado está determinado por el tipo de operación, como se muestra en la siguiente tabla:

| **Operación** | **Método HTTP** |
|---|---|
| `@ReadOperation` | `GET` |
| `@WriteOperation` | `POST` |
| `@DeleteOperation` | `DELETE` |

## Consumos

Para un `@WriteOperation` (HTTP `POST` ) que usa el cuerpo de la solicitud, la cláusula consume del predicado es `application/vnd.spring-boot.actuator.v2+json, application/json` . Para todas las demás operaciones, la cláusula consume está vacía.

## Produce

El produce cláusula del predicado puede ser determinada por el `produces` atributo de las `@DeleteOperation` , `@ReadOperation` y `@WriteOperation` las anotaciones. El atributo es opcional. Si no se usa, la cláusula produce se determina automáticamente.

Si el método de operación regresa `void` o `Void` la cláusula produce está vacía. Si el método de operación devuelve a `org.springframework.core.io.Resource` , la cláusula produce es `application/octet-stream` . Para todas las demás operaciones, la cláusula produce es `application/vnd.spring-boot.actuator.v2+json, application/json` .

## Estado de respuesta de punto final web

El estado de respuesta predeterminado para una operación de punto final depende del tipo de operación (lectura, escritura o eliminación) y de lo que devuelve la operación, si corresponde.

A `@ReadOperation` devuelve un valor, el estado de respuesta será 200 (OK). Si no devuelve un valor, el estado de respuesta será 404 (No encontrado).

Si a `@WriteOperation` o `@DeleteOperation` devuelve un valor, el estado de respuesta será 200 (OK). Si no devuelve un valor, el estado de respuesta será 204 (Sin contenido).

Si se invoca una operación sin un parámetro requerido, o con un parámetro que no se puede convertir al tipo requerido, no se llamará al método de operación y el estado de respuesta será 400 (Solicitud incorrecta).

## Solicitudes de rango de punto final web

Se puede usar una solicitud de rango HTTP para solicitar parte de un recurso HTTP. Cuando se utiliza Spring MVC o Spring Web Flux, las operaciones que devuelven `org.springframework.core.io.Resource` automáticamente admiten solicitudes de rango.

> Las solicitudes de rango no son compatibles cuando se usa Jersey.

## Seguridad de punto final web

Una operación en un punto final web o una extensión de punto final específico de la web puede recibir el parámetro actual `java.security.Principal` o `org.springframework.boot.actuate.endpoint.SecurityContext` como un método. El primero se usa generalmente junto con `@Nullable` para proporcionar un comportamiento diferente para

usuarios autenticados y no autenticados. Este último se usa generalmente para realizar verificaciones de autorización utilizando su `isUserInRole(String)` método.

### 2.7.3. Puntos finales de servlet

A `Servlet` puede exponerse como un punto final implementando una clase anotada con `@ServletEndpoint` eso que también implementa `Supplier<EndpointServlet>` . Los puntos finales de Servlet proporcionan una integración más profunda con el contenedor de Servlet pero a expensas de la portabilidad. Están destinados a ser utilizados para exponer un existente `Servlet` como punto final. Para los puntos finales nuevos, se deben preferir las anotaciones `@Endpoint` y `@WebEndpoint` siempre que sea posible.

### 2.7.4. Puntos finales del controlador

`@ControllerEndpoint` y `@RestControllerEndpoint` puede usarse para implementar un punto final que solo está expuesto por Spring MVC o Spring WebFlux. Los métodos se asignan utilizando las anotaciones estándar para Spring MVC y Spring WebFlux, como `@RequestMapping` y `@GetMapping` , con el ID del punto final como prefijo para la ruta. Los puntos finales del controlador proporcionan una integración más profunda con los marcos web de Spring pero a expensas de la portabilidad. Se deben preferir las anotaciones `@Endpoint` y `@WebEndpoint` siempre que sea posible.

## 2.8. Información de salud

Puede usar la información de salud para verificar el estado de su aplicación en ejecución. A menudo lo usa el software de monitoreo para alertar a alguien cuando un sistema de producción falla. La información expuesta por el `health` punto final depende de las propiedades `management.endpoint.health.show-details` y `management.endpoint.health.show-components` que se pueden configurar con uno de los siguientes valores:

| Nombre | Descripción |
|---|---|
| `never` | Los detalles nunca se muestran. |
| `when-authorized` | Los detalles solo se muestran a usuarios autorizados. Los roles autorizados se pueden configurar mediante `management.endpoint.health.roles` . |
| `always` | Los detalles se muestran a todos los usuarios. |

El valor por defecto es `never`. Se considera que un usuario está autorizado cuando está en uno o más de los roles del punto final. Si el punto final no tiene roles configurados (el valor predeterminado), todos los usuarios autenticados se consideran autorizados. Los roles se pueden configurar con la `management.endpoint.health.roles` propiedad

> Si ha asegurado su aplicación y desea utilizarla `always`, su configuración de seguridad debe permitir el acceso al punto final de mantenimiento tanto para usuarios autenticados como no autenticados.

La información de salud se recopila del contenido de a `HealthContributorRegistry` (de forma predeterminada, todas las `HealthContributor` instancias definidas en su `ApplicationContext`). Spring Boot incluye una serie de configuraciones automáticas `HealthContributors` y también puedes escribir la tuya.

A `HealthContributor` puede ser a `HealthIndicator` o a `CompositeHealthContributor`. A `HealthIndicator` proporciona información de salud real, incluyendo a `Status`. A `CompositeHealthContributor` proporciona un compuesto de otro `HealthContributors`. En conjunto, los contribuyentes forman una estructura de árbol para representar el estado general del sistema.

De manera predeterminada, el estado final del sistema se deriva de un `StatusAggregator` que clasifica los estados de cada uno `HealthIndicator` según una lista ordenada de estados. El primer estado de la lista ordenada se utiliza como estado general de salud. Si no `HealthIndicator` devuelve un estado conocido por el `StatusAggregator`, `UNKNOWN` se utiliza un estado.

> Se `HealthContributorRegistry` puede usar para registrar y cancelar el registro de indicadores de salud en tiempo de ejecución.

## 2.8.1. Indicadores de salud configurados automáticamente

`HealthIndicators` Spring Boot configura automáticamente lo siguiente cuando corresponde:

| Nombre | Descripción |
|---|---|
| | |

| Nombre | Descripción |
|---|---|
| `CassandraHealthIndicator` | Comprueba que una base de datos Cassandra está activa. |
| `CouchbaseHealthIndicator` | Comprueba que un clúster Couchbase está activo. |
| `DiskSpaceHealthIndicator` | Comprueba si hay poco espacio en disco. |
| `ElasticSearchRestHealthContributorAutoConfiguration` | Comprueba que un clúster Elasticsearch esté activo. |
| `HazelcastHealthIndicator` | Comprueba que un servidor Hazelcast está activo. |
| `InfluxDbHealthIndicator` | Comprueba que un servidor InfluxDB está activo. |
| `InfluxDbHealthIndicator` | Comprueba que un servidor InfluxDB está activo. |
| `JmsHealthIndicator` | Comprueba que un corredor JMS está activo. |
| `MailHealthIndicator` | Comprueba que hay un servidor de correo activo. |
| `MongoHealthIndicator` | Comprueba que una base de datos Mongo está activa. |
| `PingHealthIndicator` | Siempre responde con `UP`. |
| `RabbitHealthIndicator` | Comprueba que un servidor Rabbit está activo. |
| `RedisHealthIndicator` | Comprueba que un servidor Redis está activo. |
| `SolrHealthIndicator` | Comprueba que un servidor Solr está activo. |

Puede deshabilitarlos a todos configurando la `management.health.defaults.enabled` propiedad.

## 2.8.2. Escribir indicadores de salud personalizados

Para proporcionar información de salud personalizada, puede registrar Spring beans que implementan la `HealthIndicator` interfaz. Debe proporcionar una implementación del `health()` método y devolver una `Health` respuesta. La `Health` respuesta debe incluir un estado

y, opcionalmente, puede incluir detalles adicionales para mostrar. El siguiente código muestra una `HealthIndicator` implementación de muestra :

JAVA

```java
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();
    }

}
```

El identificador para un dado `HealthIndicator` es el nombre del bean sin el `HealthIndicator` sufijo, si existe. En el ejemplo anterior, la información de salud está disponible en una entrada denominada `my` .

Además de los `Status` tipos predefinidos de Spring Boot , también es posible `Health` devolver una costumbre `Status` que representa un nuevo estado del sistema. En tales casos, `StatusAggregator` también debe proporcionarse una implementación personalizada de la interfaz, o la implementación predeterminada debe configurarse utilizando la `management.endpoint.health.status.order` propiedad de configuración.

Por ejemplo, suponga que se está utilizando un `Status` código nuevo `FATAL` en una de sus `HealthIndicator` implementaciones. Para configurar el orden de gravedad, agregue la siguiente propiedad a las propiedades de su aplicación:

PROPERTIES

```properties
management.endpoint.health.status.order=fatal,down,out-of-service,unknown,up
```

El código de estado HTTP en la respuesta refleja el estado general de salud (por ejemplo, `UP` asigna a 200, mientras que `OUT_OF_SERVICE` y asigna `DOWN` a 503). También es posible que desee registrar asignaciones de estado personalizadas si accede al punto final de salud a través de HTTP. Por ejemplo, la siguiente propiedad se asigna `FATAL` a 503 (servicio no disponible):

PROPERTIES

```
management.endpoint.health.status.http-mapping.fatal=503
```

Si necesita más control, puede definir su propio `HttpCodeStatusMapper` bean.

La siguiente tabla muestra las asignaciones de estado predeterminadas para los estados integrados:

| Estado | Cartografía |
|---|---|
| ABAJO | SERVICIO_NO DISPONIBLE (503) |
| FUERA DE SERVICIO | SERVICIO_NO DISPONIBLE (503) |
| ARRIBA | Sin asignación por defecto, por lo que el estado http es 200 |
| DESCONOCIDO | Sin asignación por defecto, por lo que el estado http es 200 |

## 2.8.3. Indicadores de salud reactiva

Para aplicaciones reactivas, como las que usan Spring WebFlux, `ReactiveHealthContributor` proporciona un contrato sin bloqueo para obtener el estado de la aplicación. Similar a una información tradicional `HealthContributor` , la salud se recopila del contenido de un `ReactiveHealthContributorRegistry` (por defecto, todos `HealthContributor` y las `ReactiveHealthContributor` instancias definidas en su `ApplicationContext` ). Los regulares `HealthContributors` que no se comparan con una API reactiva se ejecutan en el programador elástico.

En una aplicación reactiva, `ReactiveHealthContributorRegistry` se puede usar para registrar y anular el registro de los indicadores de salud en tiempo de ejecución.

Para proporcionar información de salud personalizada de una API reactiva, puede registrar Spring Beans que implementan la `ReactiveHealthIndicator` interfaz. El siguiente código muestra una `ReactiveHealthIndicator` implementación de muestra :

JAVA

```
@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {

    @Override
    public Mono<Health> health() {
        return doHealthCheck() //perform some specific health check that returns a Mono<Healt
            .onErrorResume(ex -> Mono.just(new Health.Builder().down(ex).build()));
    }

}
```

Para manejar el error automáticamente, considere extender desde `AbstractReactiveHealthIndicator` .

## 2.8.4. Indicadores reactivos de salud configurados automáticamente

`ReactiveHealthIndicators` Spring Boot configura automáticamente lo siguiente cuando corresponde:

| Nombre | Descripción |
|---|---|
| `CassandraReactiveHealthIndicator` | Comprueba que una base de datos Cassandra está activa. |
| `CouchbaseReactiveHealthIndicator` | Comprueba que un clúster Couchbase está activo. |
| `MongoReactiveHealthIndicator` | Comprueba que una base de datos Mongo está activa. |

| Nombre | Descripción |
| --- | --- |
| `RedisReactiveHealthIndicator` | Comprueba que un servidor Redis está activo. |

Si es necesario, los indicadores reactivos reemplazan a los regulares. Además, todo lo `HealthIndicator` que no se maneja explícitamente se ajusta automáticamente.

## 2.8.5. Grupos de salud

A veces es útil organizar los indicadores de salud en grupos que se pueden usar para diferentes propósitos. Por ejemplo, si implementa su aplicación en Kubernetes, es posible que desee un conjunto diferente de indicadores de salud para sus sondas de "vitalidad" y "preparación".

Para crear un grupo de indicadores de salud, puede usar la `management.endpoint.health.group.<name>` propiedad y especificar una lista de ID de indicadores de salud para `include` o `exclude`. Por ejemplo, para crear un grupo que incluya solo indicadores de la base de datos, puede definir lo siguiente:

PROPERTIES

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

Luego puede verificar el resultado presionando `localhost:8080/actuator/health/custom`.

Por grupos predeterminados heredarán la misma `StatusAggregator` y `HttpCodeStatusMapper` configuración que el sistema de salud, sin embargo, éstos también pueden definirse en función de cada grupo. También es posible anular las propiedades `show-details` y `roles` si es necesario:

PROPERTIES

```
management.endpoint.health.group.custom.show-details=when-authorized
management.endpoint.health.group.custom.roles=admin
management.endpoint.health.group.custom.status.order=fatal,up
management.endpoint.health.group.custom.status.http-mapping.fatal=500
```

Puede usarlo `@Qualifier("groupname")` si necesita registrar personalizadas

`StatusAggregator` o `HttpCodeStatusMapper` beans para usar con el grupo.

## 2.9. Informacion de aplicacion

La información de la aplicación expone información diversa recopilada de todos los `InfoContributor` beans definidos en su `ApplicationContext` . Spring Boot incluye una cantidad de `InfoContributor` beans configurados automáticamente , y usted puede escribir los suyos.

### 2.9.1 InfoContributors configurados automáticamente

`InfoContributor` Spring Boot configura automáticamente los siguientes beans, cuando corresponde:

| Nombre | Descripción |
|---|---|
| `EnvironmentInfoContributor` | Expone cualquier clave de `Environment` debajo de la `info` clave. |
| `GitInfoContributor` | Expone información de git si hay un `git.properties` archivo disponible. |
| `BuildInfoContributor` | Expone información de compilación si hay un `META-INF/build-info.properties` archivo disponible. |

Es posible deshabilitarlos todos configurando la `management.info.defaults.enabled` propiedad.

### 2.9.2 Información de aplicación personalizada

Puede personalizar los datos expuestos por el `info` punto final configurando las `info.*` propiedades de Spring. Todas las `Environment` propiedades bajo la `info` clave se exponen automáticamente. Por ejemplo, puede agregar la siguiente configuración a su `application.properties` archivo:

```
PROPERTIES
info.app.encoding=UTF-8
info.app.java.source=1.8
info.app.java.target=1.8
```

En lugar de codificar esos valores, también podría expandir las propiedades de información en el momento de la compilación .

Suponiendo que use Maven, podría reescribir el ejemplo anterior de la siguiente manera:

```
PROPERTIES
info.app.encoding=@project.build.sourceEncoding@
info.app.java.source=@java.version@
info.app.java.target=@java.version@
```

## 2.9.3 Git Commit Information

Otra característica útil del `info` punto final es su capacidad para publicar información sobre el estado de su `git` repositorio de código fuente cuando se creó el proyecto. Si un `GitProperties` frijol está disponible, los `git.branch` , `git.commit.id` y `git.commit.time` propiedades están expuestos.

Un `GitProperties` bean se configura automáticamente si hay un `git.properties` archivo disponible en la raíz del classpath. Consulte " Generar información de git " para obtener más detalles.

Si desea mostrar la información completa de git (es decir, el contenido completo de `git.properties` ), use la `management.info.git.mode` propiedad de la siguiente manera:

```
PROPERTIES
management.info.git.mode=full
```

## 2.9.4 Información de compilación

Si hay un `BuildProperties` bean disponible, el `info` punto final también puede publicar información sobre su compilación. Esto sucede si hay un `META-INF/build-info.properties` archivo disponible en el classpath.

Los complementos Maven y Gradle pueden generar ese archivo. Consulte " Generar información de compilación " para obtener más detalles.

### 2.9.5. Escribir información personalizada

Para proporcionar información de aplicación personalizada, puede registrar Spring Beans que implementan la `InfoContributor` interfaz.

El siguiente ejemplo aporta una `example` entrada con un solo valor:

JAVA

```
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"));
    }

}
```

Si llega al `info` punto final, debería ver una respuesta que contiene la siguiente entrada adicional:

JSON

```
{
    "example": {
        "key" : "value"
    }
}
```

# 3. Monitoreo y gestión a través de HTTP

Si está desarrollando una aplicación web, Spring Boot Actuator configura automáticamente todos los puntos finales habilitados para exponerlos a través de HTTP. La convención predeterminada es usar el `id` punto final con un prefijo `/actuator` como la ruta URL. Por ejemplo, `health` se expone como `/actuator/health` .

El actuador es compatible de forma nativa con Spring MVC, Spring WebFlux y Jersey.

## 3.1. Personalizar las rutas de punto final de administración

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:

PROPERTIES

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

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

Unless the management port has been configured to expose endpoints by using a different HTTP 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 to `management.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**

PROPERTIES

```
management.endpoints.web.base-path=/
management.endpoints.web.path-mapping.health=healthcheck
```

## 3.2. Personalizar el puerto del servidor de administración

Exponer los puntos finales de administración mediante el uso del puerto HTTP predeterminado es una opción sensata para las implementaciones basadas en la nube. Sin embargo, si su aplicación se ejecuta dentro de su propio centro de datos, es posible que prefiera exponer los puntos finales utilizando un puerto HTTP diferente.

Puede configurar la `management.server.port` propiedad para cambiar el puerto HTTP, como se muestra en el siguiente ejemplo:

PROPERTIES

```
management.server.port=8081
```

> En Cloud Foundry, las aplicaciones solo reciben solicitudes en el puerto 8080 para enrutamiento HTTP y TCP, de forma predeterminada. Si desea utilizar un puerto de administración personalizado en Cloud Foundry, deberá configurar explícitamente las rutas de la aplicación para reenviar el tráfico al puerto personalizado.

## 3.3. Configurar SSL específico para la administración

Cuando se configura para usar un puerto personalizado, el servidor de administración también se puede configurar con su propio SSL mediante el uso de varias `management.server.ssl.*` propiedades. Por ejemplo, hacerlo permite que un servidor de administración esté disponible a través de HTTP mientras la aplicación principal usa HTTPS, como se muestra en la siguiente configuración de propiedades:

PROPERTIES

```
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
```

Alternativamente, tanto el servidor principal como el servidor de administración pueden usar SSL pero con diferentes almacenes de claves, de la siguiente manera:

PROPERTIES

```
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
```

## 3.4. Personalizar la dirección del servidor de administración

Puede personalizar la dirección en la que están disponibles los puntos finales de administración configurando la `management.server.address` propiedad. Hacerlo puede ser útil si desea escuchar solo en una red interna u orientada a operaciones o escuchar solo conexiones desde `localhost` .

> Puede escuchar en una dirección diferente solo cuando el puerto difiere del puerto del servidor principal.

El siguiente ejemplo `application.properties` no permite conexiones de administración remota:

PROPERTIES

```
management.server.port=8081
management.server.address=127.0.0.1
```

## 3.5. Deshabilitar puntos finales HTTP

Si no desea exponer puntos finales a través de HTTP, puede establecer el puerto de administración en `-1` , como se muestra en el siguiente ejemplo:

PROPERTIES

```
management.server.port=-1
```

Esto también se puede lograr utilizando la `management.endpoints.web.exposure.exclude` propiedad, como se muestra en el siguiente ejemplo:

PROPERTIES

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

# 4. Monitoreo y gestión sobre JMX

Las Extensiones de administración de Java (JMX) proporcionan un mecanismo estándar para monitorear y administrar aplicaciones. De manera predeterminada, esta función no está habilitada y se puede activar estableciendo la propiedad de configuración `spring.jmx.enabled` en `true` . Spring Boot expone los puntos finales de administración como MBeans JMX bajo el `org.springframework.boot` dominio de forma predeterminada.

## 4.1. Personalizar nombres MBean

El nombre del MBean generalmente se genera a partir `id` del punto final. Por ejemplo, el `health` punto final se expone como `org.springframework.boot:type=Endpoint,name=Health` .

Si su aplicación contiene más de un Spring `ApplicationContext` , es posible que los nombres entren en conflicto. Para resolver este problema, puede establecer la `spring.jmx.unique-names` propiedad para `true` que los nombres MBean sean siempre únicos.

También puede personalizar el dominio JMX bajo el cual están expuestos los puntos finales. La siguiente configuración muestra un ejemplo de cómo hacerlo en `application.properties` :

PROPERTIES

```
spring.jmx.unique-names=true
management.endpoints.jmx.domain=com.example.myapp
```

## 4.2. Deshabilitar puntos finales JMX

Si no desea exponer puntos finales sobre JMX, puede establecer la `management.endpoints.jmx.exposure.exclude` propiedad en `*` , como se muestra en el siguiente ejemplo:

PROPERTIES

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

## 4.3. Usando Jolokia para JMX sobre HTTP

Jolokia es un puente JMX-HTTP que proporciona un método alternativo para acceder a los beans JMX. Para usar Jolokia, incluya una dependencia a `org.jolokia:jolokia-core`. Por ejemplo, con Maven, agregaría la siguiente dependencia:

XML

```
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
</dependency>
```

El punto final de Jolokia puede exponerse agregando `jolokia` o `*` a la `management.endpoints.web.exposure.include` propiedad. Luego puede acceder a él utilizando `/actuator/jolokia` su servidor HTTP de administración.

### 4.3.1. Personalizando Jolokia

Jolokia tiene una serie de configuraciones que tradicionalmente configuraría al establecer parámetros de servlet. Con Spring Boot, puede usar su `application.properties` archivo. Para hacerlo, prefije el parámetro con `management.endpoint.jolokia.config.`, como se muestra en el siguiente ejemplo:

PROPERTIES

```
management.endpoint.jolokia.config.debug=true
```

### 4.3.2. Desactivar Jolokia

Si usa Jolokia pero no desea que Spring Boot lo configure, establezca la `management.endpoint.jolokia.enabled` propiedad de la `false` siguiente manera:

PROPERTIES

```
management.endpoint.jolokia.enabled=false
```

---

## 5. Registradores

Spring Boot Actuator incluye la capacidad de ver y configurar los niveles de registro de su aplicación en tiempo de ejecución. Puede ver la lista completa o la configuración de un registrador individual, que se compone tanto del nivel de registro configurado explícitamente

como del nivel de registro efectivo que le da el marco de registro. Estos niveles pueden ser uno de:

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null` indica que no hay una configuración explícita.

## 5.1. Configurar un registrador

Para configurar un registrador dado, `POST` una entidad parcial al URI del recurso, como se muestra en el siguiente ejemplo:

```
JSON
{
    "configuredLevel": "DEBUG"
}
```

Para "restablecer" el nivel específico del registrador (y utilizar la configuración predeterminada en su lugar), puede pasar un valor de `null` como `configuredLevel` .

---

# 6. Métricas

Spring Boot Actuator proporciona administración de dependencias y configuración automática para Micrometer , una fachada de métricas de aplicaciones que admite numerosos sistemas de monitoreo, que incluyen:

- AppOptics

- Atlas
- Datadog
- Dynatrace
- Elástico
- Ganglios
- Grafito
- Humio
- Afluencia
- JMX
- KairosDB
- Nueva reliquia
- Prometeo
- SignalFx
- Simple (en memoria)
- StatsD
- Frente de onda

Para obtener más información sobre las capacidades de Micrometer, consulte su documentación de referencia , en particular la sección de conceptos .

## 6.1. Empezando

Spring Boot configura automáticamente un compuesto `MeterRegistry` y agrega un registro al compuesto para cada una de las implementaciones compatibles que encuentra en el classpath. Tener una dependencia `micrometer-registry-{system}` en su classpath en tiempo de ejecución es suficiente para que Spring Boot configure el registro.

La mayoría de los registros comparten características comunes. Por ejemplo, puede deshabilitar un registro particular incluso si la implementación del registro Micrometer está en el classpath. Por ejemplo, para deshabilitar Datadog:

PROPERTIES

```
management.metrics.export.datadog.enabled=false
```

Spring Boot también agregará cualquier registro autoconfigurado al registro global compuesto estático en la `Metrics` clase a menos que usted le diga explícitamente que no:

PROPERTIES

```
management.metrics.use-global-registry=false
```

Puede registrar cualquier cantidad de `MeterRegistryCustomizer` beans para configurar aún más el registro, como aplicar etiquetas comunes, antes de que cualquier medidor se registre en el registro:

JAVA

```
@Bean
MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
    return registry -> registry.config().commonTags("region", "us-east-1");
}
```

Puede aplicar personalizaciones a implementaciones de registro particulares al ser más específico sobre el tipo genérico:

JAVA

```
@Bean
MeterRegistryCustomizer<GraphiteMeterRegistry> graphiteMetricsNamingConvention() {
    return registry -> registry.config().namingConvention(MY_CUSTOM_CONVENTION);
}
```

Con esa configuración en su lugar, puede inyectar `MeterRegistry` en sus componentes y registrar métricas:

JAVA

```
@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 también configura instrumentación incorporada (es decir, `MeterBinder` implementaciones) que puede controlar mediante configuración o marcadores de anotación dedicados.

## 6.2. Sistemas de monitoreo soportados

### 6.2.1. AppOptics

De forma predeterminada, el registro de AppOptics empuja las métricas `api.appoptics.com/v1/measurements` periódicamente. Para exportar métricas a SaaS AppOptics , se debe proporcionar su token API:

PROPERTIES

```
management.metrics.export.appoptics.api-token=YOUR_TOKEN
```

### 6.2.2. Atlas

De forma predeterminada, las métricas se exportan a Atlas que se ejecutan en su máquina local. La ubicación del servidor Atlas a utilizar se puede proporcionar utilizando:

PROPERTIES

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

### 6.2.3. Datadog

El registro Datadog empuja las métricas a datadoghq periódicamente. Para exportar métricas a Datadog , se debe proporcionar su clave API:

PROPERTIES

```
management.metrics.export.datadog.api-key=YOUR_KEY
```

También puede cambiar el intervalo en el que se envían las métricas a Datadog:

PROPERTIES

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

### 6.2.4. Dynatrace

El registro de Dynatrace empuja las métricas al URI configurado periódicamente. Para exportar métricas a Dynatrace , se deben proporcionar su token API, ID de dispositivo y URI:

PROPERTIES

```
management.metrics.export.dynatrace.api-token=YOUR_TOKEN
management.metrics.export.dynatrace.device-id=YOUR_DEVICE_ID
management.metrics.export.dynatrace.uri=YOUR_URI
```

También puede cambiar el intervalo en el que se envían las métricas a Dynatrace:

PROPERTIES

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

### 6.2.5. Elástico

Por defecto, las métricas se exportan a Elastic ejecutándose en su máquina local. La ubicación del servidor Elastic a utilizar se puede proporcionar utilizando la siguiente propiedad:

PROPERTIES

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

### 6.2.6. Ganglios

Por defecto, las métricas se exportan a Ganglia que se ejecuta en su máquina local. El host del servidor Ganglia y el puerto a usar se pueden proporcionar usando:

PROPERTIES

```
management.metrics.export.ganglia.host=ganglia.example.com
management.metrics.export.ganglia.port=9649
```

### 6.2.7. Grafito

De forma predeterminada, las métricas se exportan a Graphite que se ejecuta en su máquina local. El host del servidor Graphite y el puerto a usar se pueden proporcionar usando:

PROPERTIES

```
management.metrics.export.graphite.host=graphite.example.com
management.metrics.export.graphite.port=9004
```

Micrometer proporciona un valor predeterminado `HierarchicalNameMapper` que gobierna cómo se asigna una identificación de medidor dimensional a nombres jerárquicos planos .

Para tomar el control de este comportamiento, defina `GraphiteMeterRegistry` y suministre el suyo `HierarchicalNameMapper` . Se proporcionan un autoconfigurado `GraphiteConfig` y `Clock` beans a menos que defina el suyo propio:

JAVA

```
@Bean
public GraphiteMeterRegistry graphiteMeterRegistry(GraphiteConfig config, Clock clock) {
    return new GraphiteMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

## 6.2.8. Humio

De manera predeterminada, el registro de Humio envía periódicamente métricas a cloud.humio.com . Para exportar métricas a SaaS Humio , se debe proporcionar su token API:

PROPERTIES

```
management.metrics.export.humio.api-token=YOUR_TOKEN
```

También debe configurar una o más etiquetas para identificar la fuente de datos a la que se enviarán las métricas:

PROPERTIES

```
management.metrics.export.humio.tags.alpha=a
management.metrics.export.humio.tags.bravo=b
```

## 6.2.9. Afluencia

De forma predeterminada, las métricas se exportan a Influx que se ejecuta en su máquina local. La ubicación del servidor Influx para usar se puede proporcionar usando:

PROPERTIES

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

## 6.2.10. JMX

Micrometer proporciona un mapeo jerárquico a JMX , principalmente como una forma económica y portátil de ver las métricas localmente. Por defecto, las métricas se exportan al `metrics` dominio JMX. El dominio a usar se puede proporcionar usando:

PROPERTIES

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

Micrometer proporciona un valor predeterminado `HierarchicalNameMapper` que gobierna cómo se asigna una identificación de medidor dimensional a nombres jerárquicos planos .

Para tomar el control de este comportamiento, defina `JmxMeterRegistry` y suministre el suyo `HierarchicalNameMapper` . Se proporcionan un autoconfigurado `JmxConfig` y `Clock` beans a menos que defina el suyo propio:

JAVA

```
@Bean
public JmxMeterRegistry jmxMeterRegistry(JmxConfig config, Clock clock) {
    return new JmxMeterRegistry(config, clock, MY_HIERARCHICAL_MAPPER);
}
```

## 6.2.11. KairosDB

Por defecto, las métricas se exportan a KairosDB ejecutándose en su máquina local. La ubicación del servidor KairosDB para usar se puede proporcionar usando:

PROPERTIES

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

## 6.2.12. Nueva reliquia

El nuevo registro de Relic empuja las métricas a New Relic periódicamente. Para exportar métricas a New Relic , se deben proporcionar su clave API y su ID de cuenta:

PROPERTIES

```
management.metrics.export.newrelic.api-key=YOUR_KEY
management.metrics.export.newrelic.account-id=YOUR_ACCOUNT_ID
```

También puede cambiar el intervalo en el que se envían las métricas a New Relic:

PROPERTIES

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

## 6.2.13. Prometeo

Prometheus espera raspar o sondear instancias de aplicaciones individuales para obtener métricas. Spring Boot proporciona un punto final del actuador disponible `/actuator/prometheus` para presentar un raspado de Prometheus con el formato apropiado.

> El punto final no está disponible de forma predeterminada y debe estar expuesto, consulte exposición de puntos finales para obtener más detalles.

Aquí hay un ejemplo `scrape_config` para agregar a `prometheus.yml` :

YAML

```
scrape_configs:
  - job_name: 'spring'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['HOST:PORT']
```

Para trabajos efímeros o por lotes que pueden no existir el tiempo suficiente para ser raspados, el soporte de Prometheus Pushgateway puede usarse para exponer sus métricas a Prometheus. Para habilitar el soporte de Prometheus Pushgateway, agregue la siguiente dependencia a su proyecto:

XML

```
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_pushgateway</artifactId>
</dependency>
```

Cuando la dependencia de Prometheus Pushgateway está presente en el classpath, Spring Boot configura automáticamente un `PrometheusPushGatewayManager` bean. Esto gestiona el empuje de métricas a un Prometheus Pushgateway. Se `PrometheusPushGatewayManager` puede ajustar usando las propiedades debajo de `management.metrics.export.prometheus.pushgateway` . Para la configuración avanzada, también puede proporcionar su propio `PrometheusPushGatewayManager` bean.

### 6.2.14. SignalFx

El registro SignalFx empuja las métricas a SignalFx periódicamente. Para exportar métricas a SignalFx , se debe proporcionar su token de acceso:

PROPERTIES

```
management.metrics.export.signalfx.access-token=YOUR_ACCESS_TOKEN
```

También puede cambiar el intervalo en el que se envían las métricas a SignalFx:

PROPERTIES

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

### 6.2.15. Sencillo

Micrometer se entrega con un back-end simple en memoria que se usa automáticamente como respaldo si no se configura otro registro. Esto le permite ver qué métricas se recopilan en el punto final de métricas .

El backend en memoria se deshabilita tan pronto como esté usando cualquiera de los otros backend disponibles. También puede deshabilitarlo explícitamente:

PROPERTIES

```
management.metrics.export.simple.enabled=false
```

### 6.2.16. StatsD

El registro StatsD empuja las métricas sobre UDP a un agente StatsD con entusiasmo. De manera predeterminada, las métricas se exportan a un agente StatsD que se ejecuta en su máquina local. El host y el puerto del agente StatsD para usar se pueden proporcionar usando:

PROPERTIES

```
management.metrics.export.statsd.host=statsd.example.com
management.metrics.export.statsd.port=9125
```

También puede cambiar el protocolo de línea StatsD para usar (predeterminado a Datadog):

PROPERTIES

```
management.metrics.export.statsd.flavor=etsy
```

### 6.2.17. Frente de onda

El registro de Wavefront empuja las métricas a Wavefront periódicamente. Si está exportando métricas a Wavefront directamente, debe proporcionar su token API:

PROPERTIES

```
management.metrics.export.wavefront.api-token=YOUR_API_TOKEN
```

Alternativamente, puede usar un sidecar de Wavefront o un proxy interno configurado en su entorno que reenvíe datos de métricas al host de la API de Wavefront:

PROPERTIES

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

Si publica métricas en un proxy de Wavefront (como se describe en la documentación ), el host debe estar en el `proxy://HOST:PORT` formato.

También puede cambiar el intervalo en el que se envían las métricas a Wavefront:

PROPERTIES

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

## 6.3. Métricas compatibles

Spring Boot registra las siguientes métricas principales cuando corresponde:

- Métricas JVM, informe de utilización de:
  - Varios grupos de memoria y búfer
  - Estadísticas relacionadas con la recolección de basura
  - Utilización de hilos
  - Número de clases cargadas / descargadas
- Métricas de CPU
- Métrica del descriptor de archivo
- Métricas de consumo de Kafka
- Métricas de Log4j2: registre el número de eventos registrados en Log4j2 en cada nivel

- Métricas de inicio de sesión: registre el número de eventos registrados en Logback en cada nivel
- Métricas de tiempo de actividad: informe un indicador de tiempo de actividad y un indicador fijo que representa el tiempo de inicio absoluto de la aplicación
- Métricas de Tomcat ( `server.tomcat.mbeanregistry.enabled` debe establecerse en `true` para que se registren todas las métricas de Tomcat)
- Integración de primavera métricas

### 6.3.1. Spring MVC Metrics

La configuración automática permite la instrumentación de las solicitudes manejadas por Spring MVC. Cuando `management.metrics.web.server.request.autotime.enabled` es `true` , esta instrumentación se produce para todas las solicitudes. Alternativamente, cuando se establece en `false` , puede habilitar la instrumentación agregando `@Timed` a un método de manejo de solicitudes:

JAVA

```
@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 Una clase de controlador para habilitar tiempos en cada controlador de solicitud en el controlador.

2 Un método para habilitar para un punto final individual. Esto no es necesario si lo tiene en la clase, pero puede usarse para personalizar aún más el temporizador para este punto final en particular.

3 Un método con `longTask = true` para habilitar un temporizador de tareas largas para el método. Los temporizadores de tareas largas requieren un nombre de métrica separado y se pueden apilar con un temporizador de tareas corto.

Por defecto, las métricas se generan con el nombre `http.server.requests` ,. El nombre se puede personalizar configurando la `management.metrics.web.server.request.metric-`

`name` propiedad.

Por defecto, las métricas relacionadas con Spring MVC están etiquetadas con la siguiente información:

| Etiqueta | Descripción |
|---|---|
| `exception` | Nombre de clase simple de cualquier excepción que se produjo al manejar la solicitud. |
| `method` | Método de solicitud (por ejemplo, `GET` o `POST` ) |
| `outcome` | El resultado de la solicitud se basa en el código de estado de la respuesta. 1xx es `INFORMATIONAL` , 2xx es `SUCCESS` , 3xx es `REDIRECTION` , 4xx `CLIENT_ERROR` y 5xx es `SERVER_ERROR` |
| `status` | Código de estado HTTP de la respuesta (por ejemplo, `200` o `500` ) |
| `uri` | Solicite la plantilla de URI antes de la sustitución de variables, si es posible (por ejemplo `/api/person/{id}` ) |

Para personalizar las etiquetas, proporcione una `@Bean` que implemente `WebMvcTagsProvider` .

## 6.3.2. Spring WebFlux Metrics

La configuración automática permite la instrumentación de todas las solicitudes manejadas por controladores WebFlux y controladores funcionales.

Por defecto, las métricas se generan con el nombre `http.server.requests` . Puede personalizar el nombre configurando la `management.metrics.web.server.request.metric-name` propiedad.

Por defecto, las métricas relacionadas con WebFlux están etiquetadas con la siguiente información:

| Etiqueta | Descripción |
|---|---|
| | |

| Etiqueta | Descripción |
|---|---|
| `exception` | Nombre de clase simple de cualquier excepción que se produjo al manejar la solicitud. |
| `method` | Método de solicitud (por ejemplo, `GET` o `POST` ) |
| `outcome` | El resultado de la solicitud se basa en el código de estado de la respuesta. 1xx es `INFORMATIONAL` , 2xx es `SUCCESS` , 3xx es `REDIRECTION` , 4xx `CLIENT_ERROR` y 5xx es `SERVER_ERROR` |
| `status` | Código de estado HTTP de la respuesta (por ejemplo, `200` o `500` ) |
| `uri` | Solicite la plantilla de URI antes de la sustitución de variables, si es posible (por ejemplo `/api/person/{id}` ) |

Para personalizar las etiquetas, proporcione una `@Bean` que implemente `WebFluxTagsProvider` .

## 6.3.3. Jersey Server Metrics

Auto-configuration enables the instrumentation of requests handled by the Jersey JAX-RS implementation. When `management.metrics.web.server.request.autotime.enabled` 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:

JAVA

```
@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

On a resource class to enable timings on every request handler in the resource.

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 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.request.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`.

## 6.3.4. HTTP Client Metrics

Spring Boot Actuator gestiona la instrumentación de ambos `RestTemplate` y `WebClient`. Para eso, debe inyectarse un generador autoconfigurado y usarlo para crear instancias:

- `RestTemplateBuilder` para `RestTemplate`
- `WebClient.Builder` para `WebClient`

También es posible aplicar manualmente los personalizadores responsables de esta instrumentación, a saber, `MetricsRestTemplateCustomizer` y `MetricsWebClientCustomizer` .

Por defecto, las métricas se generan con el nombre `http.client.requests` ,. El nombre se puede personalizar configurando la `management.metrics.web.client.request.metric-name` propiedad.

De forma predeterminada, las métricas generadas por un cliente instrumentado se etiquetan con la siguiente información:

| **Etiqueta** | **Descripción** |
|---|---|
| `clientName` | Porción de host de la URI |
| `method` | Método de solicitud (por ejemplo, `GET` o `POST` ) |
| `outcome` | El resultado de la solicitud se basa en el código de estado de la respuesta. 1xx es `INFORMATIONAL` , 2xx es `SUCCESS` , 3xx es `REDIRECTION` , 4xx `CLIENT_ERROR` y 5xx es `SERVER_ERROR` , de lo `UNKNOWN` contrario |
| `status` | Código de estado HTTP de la respuesta si está disponible (por ejemplo, `200` o `500` ), o `IO_ERROR` en caso de problemas de E / S, de lo `CLIENT_ERROR` contrario |
| `uri` | Solicite la plantilla de URI antes de la sustitución de variables, si es posible (por ejemplo `/api/person/{id}` ) |

Para personalizar las etiquetas, y dependiendo de su elección de cliente, puede proporcionar una `@Bean` que implemente `RestTemplateExchangeTagsProvider` o `WebClientExchangeTagsProvider` . Hay convenientes funciones estáticas en `RestTemplateExchangeTags` y `WebClientExchangeTags` .

## 6.3.5. Métricas de caché

La configuración automática permite la instrumentación de todos los correos electrónicos disponibles `Cache` al inicio con métricas con el prefijo `cache` . La instrumentación de caché

está estandarizada para un conjunto básico de métricas. Métricas adicionales específicas de caché también están disponibles.

Se admiten las siguientes bibliotecas de caché:

- Cafeína
- EhCache 2
- Hazelcast
- Cualquier implementación compatible de JCache (JSR-107)

Las métricas están etiquetadas por el nombre de la memoria caché y por el nombre del `CacheManager` derivado del nombre del bean.

> Solo los cachés que están disponibles en el inicio están vinculados al registro. Para las memorias caché creadas sobre la marcha o mediante programación después de la fase de inicio, se requiere un registro explícito. Se `CacheMetricsRegistrar` pone a disposición un bean para facilitar ese proceso.

## 6.3.6. Métricas de DataSource

La configuración automática permite la instrumentación de todos los `DataSource` objetos disponibles con métricas con el prefijo `jdbc.connections` . La instrumentación del origen de datos da como resultado indicadores que representan las conexiones actualmente activas, inactivas, máximas permitidas y mínimas permitidas en el grupo.

Las métricas también están etiquetadas por el nombre del `DataSource` cómputo basado en el nombre del bean.

> De manera predeterminada, Spring Boot proporciona metadatos para todas las fuentes de datos compatibles; puede agregar `DataSourcePoolMetadataProvider` beans adicionales si su fuente de datos favorita no es compatible de inmediato. Ver `DataSourcePoolMetadataProvidersConfiguration` para ejemplos.

Además, las métricas específicas de Hikari se exponen con un `hikaricp` prefijo. Cada métrica está etiquetada por el nombre del Pool (se puede controlar con `spring.datasource.name` ).

### 6.3.7. Métricas de hibernación

La configuración automática habilita la instrumentación de todas las `EntityManagerFactory` instancias de Hibernate disponibles que tienen estadísticas habilitadas con una métrica denominada `hibernate` .

Las métricas también están etiquetadas por el nombre del `EntityManagerFactory` derivado del nombre del bean.

Para habilitar las estadísticas, la propiedad JPA estándar `hibernate.generate_statistics` debe establecerse en `true` . Puede habilitarlo en la configuración automática `EntityManagerFactory` como se muestra en el siguiente ejemplo:

PROPERTIES

```
spring.jpa.properties.hibernate.generate_statistics=true
```

### 6.3.8. RabbitMQ Metrics

La configuración automática permitirá la instrumentación de todas las fábricas de conexión RabbitMQ disponibles con una métrica denominada `rabbitmq` .

## 6.4. Registrar métricas personalizadas

Para registrar métricas personalizadas, inyecte `MeterRegistry` en su componente, como se muestra en el siguiente ejemplo:

JAVA

```
class Dictionary {

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

    Dictionary(MeterRegistry registry) {
        registry.gaugeCollectionSize("dictionary.size", Tags.empty(), this.words);
    }

    // …

}
```

Si descubre que instrumenta repetidamente un conjunto de métricas entre componentes o aplicaciones, puede encapsular este conjunto en una `MeterBinder` implementación. De forma

predeterminada, las métricas de todos los `MeterBinder` beans se vincularán automáticamente a Spring-managed `MeterRegistry` .

## 6.5. Personalizar métricas individuales

Si necesita aplicar personalizaciones a `Meter` instancias específicas , puede usar la `io.micrometer.core.instrument.config.MeterFilter` interfaz. Por defecto, todos los `MeterFilter` beans se aplicarán automáticamente al micrómetro `MeterRegistry.Config` .

Por ejemplo, si desea cambiar el nombre de la `mytag.region` etiqueta `mytag.area` para todas las ID de medidor que comiencen `com.example` , puede hacer lo siguiente:

JAVA

```
@Bean
public MeterFilter renameRegionTagMeterFilter() {
    return MeterFilter.renameTag("com.example", "mytag.region", "mytag.area");
}
```

### 6.5.1. Etiquetas comunes

Las etiquetas comunes se utilizan generalmente para la profundización dimensional en el entorno operativo como host, instancia, región, pila, etc. Las etiquetas comunes se aplican a todos los medidores y se pueden configurar como se muestra en el siguiente ejemplo:

PROPERTIES

```
management.metrics.tags.region=us-east-1
management.metrics.tags.stack=prod
```

El ejemplo anterior agrega `region` y `stack` etiqueta a todos los medidores con un valor de `us-east-1` y `prod` respectivamente.

> El orden de las etiquetas comunes es importante si está utilizando Graphite. Como no se puede garantizar el orden de las etiquetas comunes con este enfoque, se recomienda a los usuarios de Graphite que definan una personalizada `MeterFilter` .

### 6.5.2. Propiedades por metro

Además de los `MeterFilter` beans, también es posible aplicar un conjunto limitado de personalización por metro utilizando propiedades. Las personalizaciones por medidor se

aplican a todas las ID de medidor que comienzan con el nombre dado. Por ejemplo, lo siguiente deshabilitará cualquier medidor que tenga una ID que comience con `example.remote`

PROPERTIES

```
management.metrics.enable.example.remote=false
```

Las siguientes propiedades permiten la personalización por metro:

**Tabla 1. Personalizaciones por metro**

| Propiedad | Descripción |
|---|---|
| `management.metrics.enable` | Si se debe negar a los medidores la emisión de alguna métrica. |
| `management.metrics.distribution.percentiles-histogram` | Si se debe publicar un histograma adecuado para calcular aproximaciones de percentiles agregados (a través de dimensiones). |
| `management.metrics.distribution.minimum-expected-value` , `management.metrics.distribution.maximum-expected-value` | Publique menos cubos de histograma sujetando el rango de valores esperados. |
| `management.metrics.distribution.percentiles` | Publique valores de percentiles calculados en su aplicación |
| `management.metrics.distribution.sla` | Publique un histograma acumulativo con cubos definidos por sus SLA. |

For more details on concepts behind `percentiles-histogram` , `percentiles` and `sla` refer to the "Histograms and percentiles" section of the micrometer documentation.

## 6.6. 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 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` .

> 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` .

> Las mediciones informadas son la *suma* de las estadísticas de todos los medidores que coinciden con el nombre del medidor y cualquier etiqueta que se haya aplicado. Entonces, en el ejemplo anterior, la estadística de "Valor" devuelta es la suma de las huellas de memoria máximas de las áreas de "Caché de código", "Espacio de clase comprimido" y "Metaspace" del montón. Si solo desea ver el tamaño máximo para el "Metaspace", puede agregar un adicional `tag=id:Metaspace` , es decir `/actuator/metrics/jvm.memory.max?tag=area:nonheap&tag=id:Metaspace` .

---

# 7. Auditoría

Una vez que Spring Security está en juego, Spring Boot Actuator tiene un marco de auditoría flexible que publica eventos (por defecto, excepciones de "éxito de autenticación", "falla" y "acceso denegado"). Esta característica puede ser muy útil para informar y para implementar una política de bloqueo basada en fallas de autenticación.

La auditoría se puede habilitar proporcionando un bean de tipo `AuditEventRepository` en la configuración de su aplicación. Para mayor comodidad, Spring Boot ofrece un `InMemoryAuditEventRepository` . `InMemoryAuditEventRepository` tiene capacidades limitadas y recomendamos usarlo solo para entornos de desarrollo. Para entornos de producción, considere crear su propia `AuditEventRepository` implementación alternativa .

## 7.1. Auditoria personalizada

Para personalizar los eventos de seguridad publicados, puede proporcionar sus propias implementaciones de `AbstractAuthenticationAuditListener` y `AbstractAuthorizationAuditListener` .

También puede utilizar los servicios de auditoría para sus propios eventos comerciales. Para hacerlo, inyecte el `AuditEventRepository` bean en sus propios componentes y úselo directamente o publique uno `AuditApplicationEvent` con Spring `ApplicationEventPublisher` (implementando `ApplicationEventPublisherAware` ).

---

# 8. Rastreo HTTP

El seguimiento HTTP se puede habilitar proporcionando un bean de tipo `HttpTraceRepository` en la configuración de su aplicación. Para mayor comodidad, Spring Boot ofrece un `InMemoryHttpTraceRepository` almacenamiento de rastros para los últimos 100 intercambios de solicitud-respuesta, de forma predeterminada. `InMemoryHttpTraceRepository` es limitado en comparación con otras soluciones de rastreo y recomendamos usarlo solo para entornos de desarrollo. Para entornos de producción, se recomienda el uso de una solución de rastreo o de observación lista para producción, como Zipkin o Spring Cloud Sleuth. Alternativamente, cree el suyo `HttpTraceRepository` que satisfaga sus necesidades.

El `httptrace` punto final se puede utilizar para obtener información sobre los intercambios de solicitud-respuesta que se almacenan en el `HttpTraceRepository` .

## 8.1. Seguimiento HTTP personalizado

Para personalizar los elementos que se incluyen en cada rastreo, use la `management.trace.http.include` propiedad de configuración. Para una personalización avanzada, considere registrar su propia `HttpExchangeTracer` implementación.

---

# 9. Monitoreo de procesos

En el `spring-boot` módulo, puede encontrar dos clases para crear archivos que a menudo son útiles para la supervisión de procesos:

- `ApplicationPidFileWriter` crea un archivo que contiene el PID de la aplicación (de manera predeterminada, en el directorio de la aplicación con un nombre de archivo `application.pid` ).

- `WebServerPortFileWriter` crea un archivo (o archivos) que contiene los puertos del servidor web en ejecución (de forma predeterminada, en el directorio de la aplicación con un nombre de archivo `application.port` ).

De manera predeterminada, estos escritores no están activados, pero puede habilitar:

- Al extender la configuración
- Programáticamente

## 9.1. Configuración extendida

En el `META-INF/spring.factories` archivo, puede activar los oyentes que escriben un archivo PID, como se muestra en el siguiente ejemplo:

```
org.springframework.context.ApplicationListener = \
org.springframework.boot.context.ApplicationPidFileWriter, \
org.springframework.boot.web.context.WebServerPortFileWriter
```

## 9.2. Programáticamente

También puede activar un oyente invocando el `SpringApplication.addListeners(…)` método y pasando el `Writer` objeto apropiado . Este método también le permite personalizar el nombre del archivo y la ruta en el `Writer` constructor.

---

# 10. Soporte de Cloud Foundry

El módulo actuador de Spring Boot incluye soporte adicional que se activa cuando se implementa en una instancia de Cloud Foundry compatible. La `/cloudfoundryapplication` ruta proporciona una ruta segura alternativa a todos los `@Endpoint` beans.

El soporte extendido permite que las IU de administración de Cloud Foundry (como la aplicación web que puede usar para ver las aplicaciones implementadas) se incrementen con la información del actuador Spring Boot. Por ejemplo, una página de estado de la aplicación puede incluir información completa sobre el estado en lugar del estado típico de "en ejecución" o "detenido".

La `/cloudfoundryapplication` ruta no es directamente accesible para los usuarios

habituales. Para usar el punto final, se debe pasar un token UAA válido con la solicitud.

## 10.1 Desactivación del soporte del actuador de Cloud Foundry extendido

Si desea deshabilitar completamente los `/cloudfoundryapplication` puntos finales, puede agregar la siguiente configuración a su `application.properties` archivo:

**application.properties**

PROPERTIES

```
management.cloudfoundry.enabled=false
```

## 10.2 Certificados autofirmados de Cloud Foundry

De forma predeterminada, la verificación de seguridad para `/cloudfoundryapplication` puntos finales realiza llamadas SSL a varios servicios de Cloud Foundry. Si sus servicios Cloud Foundry UAA o Cloud Controller usan certificados autofirmados, debe establecer la siguiente propiedad:

**application.properties**

PROPERTIES

```
management.cloudfoundry.skip-ssl-validation=true
```

## 10.3 Ruta de contexto personalizada

Si la ruta de contexto del servidor se ha configurado para algo diferente `/` , los puntos finales de Cloud Foundry no estarán disponibles en la raíz de la aplicación. Por ejemplo, si `server.servlet.context-path=/app` , los puntos finales de Cloud Foundry estarán disponibles en `/app/cloudfoundryapplication/*` .

Si espera que los puntos finales de Cloud Foundry estén siempre disponibles `/cloudfoundryapplication/*` , independientemente de la ruta de contexto del servidor, deberá configurarlo explícitamente en su aplicación. La configuración diferirá según el servidor web en uso. Para Tomcat, se puede agregar la siguiente configuración:

JAVA

```
@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(getContext
            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 ServletExcept
                ServletContext context = req.getServletContext().getContext(contextPath);
                context.getRequestDispatcher("/cloudfoundryapplication").forward(req, res);
            }

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

# 11. Qué leer a continuación

Si desea explorar algunos de los conceptos discutidos en este capítulo, puede echar un vistazo a las aplicaciones de muestra de actuadores . También es posible que desee leer sobre herramientas gráficas como Graphite .

De lo contrario, puede continuar para leer sobre 'opciones de implementación' o avanzar para obtener información detallada sobre los *complementos de la herramienta de compilación de* Spring Boot .

Última actualización 2019-10-16 17:33:16 UTC