(/)

Actuador de arranque de resorte

Última modificación: 21 de febrero de 2019

por José Carlos Valero Sánchez (https://www.baeldung.com/author/jose-valero/)
(https://www.baeldung.com/author/jose-valero/)

Spring Boot (https://www.baeldung.com/category/spring/spring-boot/)

Fundamentos de arranque (https://www.baeldung.com/tag/boot-basics/)

1. Información general

En este artículo, vamos a presentar Spring Boot Actuator. **Primero cubriremos los conceptos básicos, luego discutiremos en detalle lo que está disponible en Spring Boot 1.x vs 2.x.**

Aprenderemos cómo usar, configurar y extender esta herramienta de monitoreo en Spring Boot 1.x. Luego, discutiremos cómo hacer lo mismo usando Boot 2.xy WebFlux aprovechando el modelo de programación reactiva.

Spring Boot Actuator está disponible desde abril de 2014, junto con el primer lanzamiento de Spring Boot.

Con el lanzamiento de Spring Boot 2 (/new-spring-boot-2) , Actuator ha sido rediseñado y se agregaron nuevos puntos finales emocionantes.

Esta guía se divide en 3 secciones principales:

- **¿Qué es un actuador?**
- **Actuador Spring Boot 1.x**
- **Actuador Spring Boot 2.x**

Otras lecturas:

Configurar una aplicación web Spring Boot (https://www.baeldung.com/spring-boot-application-configuration)

Algunas de las configuraciones más útiles para una aplicación Spring Boot.

Leer más (https://www.baeldung.com/spring-boot-application-configuration) →

Crear un iniciador personalizado con Spring Boot (https://www.baeldung.com/spring-boot-custom-starter)

Una guía rápida y práctica para crear arrancadores Spring Boot personalizados.

Leer más (https://www.baeldung.com/spring-boot-custom-starter) →

Pruebas en Spring Boot (https://www.baeldung.com/spring-boot-testing)

Aprenda cómo Spring Boot admite pruebas, para escribir pruebas unitarias de manera eficiente.

Leer más (https://www.baeldung.com/spring-boot-testing) →

2. ¿Qué es un actuador?

En esencia, Actuator trae funciones listas para producción a nuestra aplicación.

Monitorear nuestra aplicación, recopilar métricas, comprender el tráfico o el estado de nuestra base de datos se vuelve trivial con esta dependencia.

El principal beneficio de esta biblioteca es que podemos obtener herramientas de calidad de producción sin tener que implementar estas características nosotros mismos.

Actuator se utiliza principalmente para **exponer información operativa sobre la aplicación en ejecución** : estado, métricas, información, volcado, env, etc. Utiliza puntos finales HTTP o beans JMX para permitirnos interactuar con ella.

Una vez que esta dependencia está en el classpath, hay varios puntos finales disponibles para nosotros de forma inmediata. Como con la mayoría de los módulos Spring, podemos configurarlo o extenderlo fácilmente de muchas maneras.

2.1. Empezando

Para habilitar Spring Boot Actuator solo necesitaremos agregar la dependencia de *spring-boot-actuator* a nuestro administrador de paquetes. En Maven:

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

Tenga en cuenta que esto sigue siendo válido independientemente de la versión de arranque, ya que las versiones se especifican en Spring Boot Bill of Materials (BOM).

3. Actuador Spring Boot 1.x

En 1.x Actuator sigue un modelo R / W, eso significa que podemos leerlo o escribirle. Por ejemplo, podemos recuperar métricas o el estado de nuestra aplicación. Alternativamente, podríamos finalizar con gracia nuestra aplicación o cambiar nuestra configuración de registro.

Para que funcione, Actuator requiere Spring MVC para exponer sus puntos finales a través de HTTP. Ninguna otra tecnología es compatible.

3.1. Puntos finales

En 1.x, Actuator trae su propio modelo de seguridad. Aprovecha las construcciones de Spring Security, pero debe configurarse independientemente del resto de la aplicación.

Además, la mayoría de los puntos finales son sensibles, lo que significa que no son completamente públicos, o en otras palabras, se omitirá la mayoría de la información, mientras que un puñado no es, por ejemplo, *⁄ info* .

Estos son algunos de los puntos finales más comunes que Boot proporciona de fábrica:

- *⁄ salud* : muestra información de salud de la aplicación (un *"estado"* simple cuando se accede a través de una conexión no autenticada o detalles completos del mensaje cuando se autentica); no es sensible por defecto
- *⁄ info:* muestra información de aplicación arbitraria; no sensible por defecto
- *⁄ metrics:* muestra información de 'métricas' para la aplicación actual; También es sensible por defecto
- *⁄ trace:* muestra información de seguimiento (de forma predeterminada, las últimas pocas solicitudes HTTP)

Podemos encontrar la lista completa de puntos finales existentes en los documentos oficiales (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints) .

3.2. Configurar puntos finales existentes

Cada punto final se puede personalizar con propiedades utilizando el siguiente formato: *puntos finales. [Nombre del punto final]. [Propiedad para personalizar]*

Hay tres propiedades disponibles:

- *id:* mediante el cual se accederá a este punto final a través de HTTP
- *habilitado* : si es verdadero, se puede acceder de lo contrario no
- *sensible* : si es verdadero, entonces necesita la autorización para mostrar información crucial sobre HTTP

Por ejemplo, agregar las siguientes propiedades personalizará el punto final *⁄ beans :*

```
endpoints.beans.id=springbeans
endpoints.beans.sensitive=false
endpoints.beans.enabled=true
```

3.3. ⁄ punto final de *salud*

El punto final *⁄ health* se utiliza para verificar el estado o la salud de la aplicación en ejecución. Por lo general, lo ejerce el software de monitoreo para alertarnos si la instancia en ejecución se cae o no es saludable por otras razones. Por ejemplo, problemas de conectividad con nuestra base de datos, falta de espacio en disco ...

Por defecto, solo la información de salud se muestra con acceso no autorizado a través de HTTP:

```
{
    "status" : "UP"
}
```

Esta información de salud se recopila de todos los beans que implementan la interfaz *HealthIndicator* configurada en nuestro contexto de aplicación.

Parte de la información devuelta por *HealthIndicator* es de naturaleza confidencial, pero podemos configurar *endpoints.health.sensitive = false* para exponer información más detallada como espacio en disco, conectividad de intermediario de mensajería, comprobaciones personalizadas, etc.

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

```
@Component
public class HealthCheck implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down()
              .withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

    public int check() {
        // Our logic to check health
        return 0;
    }
}
```

Here's how the output would look like:

```
{
    "status" : "DOWN",
    "myHealthCheck" : {
        "status" : "DOWN",
        "Error Code" : 1
     },
     "diskSpace" : {
         "status" : "UP",
         "free" : 209047318528,
         "threshold" : 10485760
     }
}
```

3.4. */info* Endpoint

We can also customize the data shown by the */info* endpoint – for example:

```
info.app.name=Spring Sample Application
info.app.description=This is my first spring boot application
info.app.version=1.0.0
```

And the sample output:

```
{
    "app" : {
        "version" : "1.0.0",
        "description" : "This is my first spring boot application",
        "name" : "Spring Sample Application"
    }
}
```

3.5. */metrics* Endpoint

The metrics endpoint publishes information about OS, JVM as well as application level metrics. Once enabled, we get information such as memory, heap, processors, threads, classes loaded, classes unloaded, thread pools along with some HTTP metrics as well.

Here's what the output of this endpoint looks like out of the box:

```
{
    "mem" : 193024,
    "mem.free" : 87693,
    "processors" : 4,
    "instance.uptime" : 305027,
    "uptime" : 307077,
    "systemload.average" : 0.11,
    "heap.committed" : 193024,
    "heap.init" : 124928,
    "heap.used" : 105330,
    "heap" : 1764352,
    "threads.peak" : 22,
    "threads.daemon" : 19,
    "threads" : 22,
    "classes" : 5819,
    "classes.loaded" : 5819,
    "classes.unloaded" : 0,
    "gc.ps_scavenge.count" : 7,
    "gc.ps_scavenge.time" : 54,
    "gc.ps_marksweep.count" : 1,
    "gc.ps_marksweep.time" : 44,
    "httpsessions.max" : -1,
    "httpsessions.active" : 0,
    "counter.status.200.root" : 1,
    "gauge.response.root" : 37.0
}
```

In order to gather custom metrics, we have support for 'gauges', that is, single value snapshots of data, and 'counters' i.e. incrementing/decrementing metrics.

Let's implement our own custom metrics into the */metrics* endpoint. For example, we'll customize the login flow to record a successful and failed login attempt:

```
@Service
public class LoginServiceImpl {

    private final CounterService counterService;

    public LoginServiceImpl(CounterService counterService) {
        this.counterService = counterService;
    }

    public boolean login(String userName, char[] password) {
        boolean success;
        if (userName.equals("admin") && "secret".toCharArray().equals(password)) {
            counterService.increment("counter.login.success");
            success = true;
        }
        else {
            counterService.increment("counter.login.failure");
            success = false;
        }
        return success;
    }
}
```

Here's what the output might look like:

```
{
    ...
    "counter.login.success" : 105,
    "counter.login.failure" : 12,
    ...
}
```

Note that login attempts and other security related events are available out of the box in Actuator as audit events.

3.6. Creating A New Endpoint

Besides using the existing endpoints provided by Spring Boot, we could also create an entirely new one.

Firstly, we'd need to have the new endpoint implement the *Endpoint<T>* interface:

```
@Component
public class CustomEndpoint implements Endpoint<List<String>> {

    @Override
    public String getId() {
        return "customEndpoint";
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    @Override
    public boolean isSensitive() {
        return true;
    }

    @Override
    public List<String> invoke() {
        // Custom logic to build the output
        List<String> messages = new ArrayList<String>();
        messages.add("This is message 1");
        messages.add("This is message 2");
        return messages;
    }
}
```

Para acceder a este nuevo punto final, su *ID* se usa para *mapearlo* , es decir, podríamos ejercitarlo *presionando / customEndpoint* .

Salida:

```
[ "This is message 1", "This is message 2" ]
```

3.7. Personalización adicional

Por motivos de seguridad, podríamos elegir exponer los puntos finales del actuador a través de un puerto no estándar; la propiedad *management.port* se puede usar fácilmente para configurar eso.

Además, como ya mencionamos, en 1.x. Actuator configura su propio modelo de seguridad, basado en Spring Security pero independiente del resto de la aplicación.
Por lo tanto, podemos cambiar la propiedad *management.address* para restringir dónde se puede acceder a los puntos finales desde la red:

```
#port used to expose actuator
management.port=8081

#CIDR allowed to hit actuator
management.address=127.0.0.1

#Whether security should be enabled or disabled altogether
management.security.enabled=false
```

Además, todos los puntos finales integrados excepto */ info* son sensibles por defecto. Si la aplicación usa Spring Security, podemos asegurar estos puntos finales definiendo las propiedades de seguridad predeterminadas (nombre de usuario, contraseña y rol) en el archivo application.properties:

```
security.user.name=admin
security.user.password=secret
management.security.role=SUPERUSER
```

4. Actuador Spring Boot 2.x

En 2.x, Actuator mantiene su intención fundamental, pero simplifica su modelo, extiende sus capacidades e incorpora mejores valores predeterminados.

En primer lugar, esta versión se convierte en tecnología independiente. Además, simplifica su modelo de seguridad al fusionarlo con el de la aplicación.

Por último, entre los diversos cambios, es importante tener en cuenta que algunos de ellos se están rompiendo. Esto incluye las solicitudes / respuestas HTTP, así como las API de Java.

Además, la última versión admite ahora el modelo CRUD, a diferencia del antiguo modelo RW (lectura / escritura).

4.1. Soporte tecnológico

Con su segunda versión principal, Actuator ahora es independiente de la tecnología, mientras que en 1.x estaba vinculado a MVC, por lo tanto, a la API de Servlet.

En 2.x, Actuator define su modelo, conectable y extensible sin depender de MVC para esto.
Por lo tanto, con este nuevo modelo, podemos aprovechar MVC y WebFlux como tecnología web subyacente.

Además, se podrían agregar tecnologías futuras implementando los adaptadores adecuados.

Por último, JMX sigue siendo compatible para exponer puntos finales sin ningún código adicional.

4.2. Cambios importantes

A diferencia de las versiones anteriores, **Actuator viene con la mayoría de los puntos finales deshabilitados** .

Por lo tanto, los únicos dos disponibles por defecto son */ health* y */ info* .

Si quisiéramos habilitarlos a todos, podríamos establecer *management.endpoints.web.exposure.include = *.* Alternativamente, podríamos enumerar los puntos finales que deberían estar habilitados.

Actuator ahora comparte la configuración de seguridad con las reglas normales de seguridad de la aplicación. Por lo tanto, el modelo de seguridad se simplifica dramáticamente.

Por lo tanto, para ajustar las reglas de seguridad de Actuator, podríamos agregar una entrada para */ actuator / ***:

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
      .pathMatchers("/actuator/**").permitAll()
      .anyExchange().authenticated()
      .and().build();
}
```

Podemos encontrar más detalles sobre los nuevos documentos oficiales de Actuator (https://docs.spring.io/spring-boot/docs/2.0.x/actuator-api/html/) .

Además, **por defecto, todos los puntos finales del Actuador ahora se colocan bajo la ruta */ actuador .**

Igual que en la versión anterior, podemos ajustar esta ruta, utilizando la nueva propiedad *management.endpoints.web.base-path.*

4.3. Puntos finales predefinidos

Echemos un vistazo a algunos puntos finales disponibles, la mayoría de ellos ya estaban disponibles en 1.x.

No obstante, **se agregaron algunos puntos finales, se eliminaron algunos y se reestructuraron algunos:**

- */ auditevents:* enumera los eventos relacionados con la auditoría de seguridad, como el inicio / cierre de sesión del usuario. Además, podemos filtrar por principal o tipo entre otros campos
- */ beans:* devuelve todos los beans disponibles en nuestra *BeanFactory* . A diferencia de */ auditevents* , no admite el filtrado
- */ condiciones:* anteriormente conocido como */ autoconfig* , crea un informe de condiciones en torno a la configuración automática
- */ configprops:* nos permite obtener todos los beans de *@ConfigurationProperties*
- */ env:* devuelve las propiedades del entorno actual. Además, podemos recuperar propiedades individuales
- */ flyway:* proporciona detalles sobre las migraciones de nuestra base de datos Flyway
- */ salud:* resume el estado de salud de nuestra aplicación
- */ heapdump:* crea y devuelve un volcado de pila desde la JVM utilizada por nuestra aplicación
- */ info:* devuelve información general. Pueden ser datos personalizados, información de compilación o detalles sobre la última confirmación
- */ liquibase - b* tiene como */ flyway* pero para Liquibase
- */ logfile:* devuelve registros de aplicaciones normales

- */ loggers:* nos permite consultar y modificar el nivel de registro de nuestra aplicación
- */ metrics:* detalla las métricas de nuestra aplicación. Esto podría incluir métricas genéricas y personalizadas.
- */ prometheus:* devuelve métricas como la anterior, pero formateadas para funcionar con un servidor Prometheus
- */ scheduletasks:* proporciona detalles sobre cada tarea programada dentro de nuestra aplicación
- */ sessions:* enumera las sesiones HTTP dado que estamos usando Spring Session
- */ shutdown:* realiza un apagado correcto de la aplicación
- */ threaddump:* vuelca la información del hilo de la JVM subyacente

4.4. Indicadores de salud

Al igual que en la versión anterior, podemos agregar indicadores personalizados fácilmente. Frente a otras API, las abstracciones para crear puntos finales de salud personalizados permanecen sin cambios. Sin embargo, **se ha agregado una nueva interfaz *ReactiveHealthIndicator* para implementar controles de salud reactivos** .

Echemos un vistazo a un simple control de salud reactiva personalizado:

```
@Component
public class DownstreamServiceHealthIndicator implements ReactiveHealthIndicator {

    @Override
    public Mono<Health> health() {
        return checkDownstreamServiceHealth().onErrorResume(
          ex -> Mono.just(new Health.Builder().down(ex).build())
        );
    }

    private Mono<Health> checkDownstreamServiceHealth() {
        // we could use WebClient to check health reactively
        return Mono.just(new Health.Builder().up().build());
    }
}
```

Una característica útil de los indicadores de salud es que podemos agregarlos como parte de una jerarquía. Por lo tanto, siguiendo el ejemplo anterior, podríamos agrupar todos los servicios derivados bajo un *downstream- servicios de* categoría. Esta categoría sería válida siempre que se pudiera acceder a todos los *servicios* anidados .

Las comprobaciones de estado compuestas están presentes en 1.x a través de *CompositeHealthIndicator.* Además, en 2.x podríamos usar *CompositeReactiveHealthIndicator* para su contraparte reactiva.

A diferencia de Spring Boot 1.x, los *puntos finales.* <id>. *La* bandera *sensible* ha sido eliminada. Para ocultar el informe de salud completo, podemos aprovechar la nueva *gestión.endpoint.health.show-details.* Esta bandera es falsa por defecto.

4.5. Métricas en Spring Boot 2

En Spring Boot 2.0, las métricas internas fueron reemplazadas por soporte de micrómetro. Por lo tanto, podemos esperar cambios importantes. Si nuestra aplicación usaba servicios métricos como *GaugeService o CounterService* , ya no estarán disponibles.

En cambio, se espera que interactuemos directamente con Micrometer (/micrometer) . En Spring Boot 2.0, obtendremos un bean de tipo *MeterRegistry* autoconfigurado para nosotros.

Además, Micrometer ahora es parte de las dependencias de Actuator. Por lo tanto, deberíamos ser buenos siempre que la dependencia del Actuador esté en el classpath.

Además, obtendremos una respuesta completamente nueva desde el punto final */ metrics :*

```
{
  "names": [
    "jvm.gc.pause",
    "jvm.buffer.memory.used",
    "jvm.memory.used",
    "jvm.buffer.count",
    // ...
  ]
}
```

Como podemos observar en el ejemplo anterior, no hay métricas reales como obtuvimos en 1.x.

Para obtener el valor real de una métrica específica, ahora podemos navegar a la métrica deseada, es decir, */actuator/metrics/jvm.gc.pause* y obtener una respuesta detallada:

```
{
  "name": "jvm.gc.pause",
  "measurements": [
    {
      "statistic": "Count",
      "value": 3.0
    },
    {
      "statistic": "TotalTime",
      "value": 7.9E7
    },
    {
      "statistic": "Max",
      "value": 7.9E7
    }
  ],
  "availableTags": [
    {
      "tag": "cause",
      "values": [
        "Metadata GC Threshold",
        "Allocation Failure"
      ]
    },
    {
      "tag": "action",
      "values": [
        "end of minor GC",
        "end of major GC"
      ]
    }
  ]
}
```

Como podemos ver, las métricas ahora son mucho más exhaustivas. Incluyendo no solo valores diferentes sino también algunos metadatos asociados.

4.6. Personalizar el punto final */ info*

El punto final */ info* permanece sin cambios. **Como antes, podemos agregar detalles de git usando la dependencia respectiva de Maven o Gradle** :

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

Del mismo modo, **también podríamos incluir información de compilación que incluya nombre, grupo y versión utilizando el complemento Maven o Gradle:**

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

4.7. Crear un punto final personalizado

Como señalamos anteriormente, podemos crear puntos finales personalizados. Sin embargo, Spring Boot 2 ha rediseñado la forma de lograr esto para soportar el nuevo paradigma agnóstico de tecnología.

Creemos un punto final de Actuator para consultar, habilitar y deshabilitar indicadores de características en nuestra aplicación :

```
@Component
@Endpoint(id = "features")
public class FeaturesEndpoint {

    private Map<String, Feature> features = new ConcurrentHashMap<>();

    @ReadOperation
    public Map<String, Feature> features() {
        return features;
    }

    @ReadOperation
    public Feature feature(@Selector String name) {
        return features.get(name);
    }

    @WriteOperation
    public void configureFeature(@Selector String name, Feature feature) {
        features.put(name, feature);
    }

    @DeleteOperation
    public void deleteFeature(@Selector String name) {
        features.remove(name);
    }

    public static class Feature {
        private Boolean enabled;

        // [...] getters and setters
    }

}
```

Para obtener el punto final, necesitamos un bean. En nuestro ejemplo, estamos usando *@Component* para esto. Además, necesitamos decorar este bean con *@Endpoint* .

La ruta de nuestro punto final está determinada por el parámetro *id* de *@Endpoint* , en nuestro caso, *enrutará las* solicitudes a */ actuator / features.*

Una vez listo, podemos comenzar a definir operaciones usando:

- *@ReadOperation:* se *asignará* a HTTP *GET*
- *@WriteOperation* : se *asignará* a HTTP *POST*
- *@DeleteOperation* : se *asignará* a HTTP *DELETE*

Cuando ejecutamos la aplicación con el punto final anterior en nuestra aplicación, Spring Boot la registrará.

Una forma rápida de verificar esto sería verificar los registros:

```
[...].WebFluxEndpointHandlerMapping: Mapped "{[/actuator/features/{name}],
  methods=[GET],
  produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}"
[...].WebFluxEndpointHandlerMapping : Mapped "{[/actuator/features],
  methods=[GET],
  produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}"
[...].WebFluxEndpointHandlerMapping : Mapped "{[/actuator/features/{name}],
  methods=[POST],
  consumes=[application/vnd.spring-boot.actuator.v2+json || application/json]}"
[...].WebFluxEndpointHandlerMapping : Mapped "{[/actuator/features/{name}],
  methods=[DELETE]}"[...]
```

En los registros anteriores, podemos ver cómo WebFlux está exponiendo nuestro nuevo punto final. ¿Cambiaríamos a MVC? Simplemente delegará en esa tecnología sin tener que cambiar ningún código.

Además, tenemos algunas consideraciones importantes a tener en cuenta con este nuevo enfoque:

- No hay dependencias con MVC
- Todos los metadatos presentes como métodos anteriores (*sensibles, habilitados ...)* ya no existen. Sin embargo, podemos habilitar o deshabilitar el punto final usando *@Endpoint (id = "features", enableByDefault = false)*
- A diferencia de 1.x, ya no es necesario extender una interfaz determinada
- A diferencia del antiguo modelo de lectura / escritura, ahora podemos definir operaciones *DELETE* usando *@DeleteOperation*

4.8. Extensión de puntos finales existentes

Imaginemos que queremos asegurarnos de que la instancia de producción de nuestra aplicación nunca sea una versión *SNAPSHOT* . Decidimos hacer esto cambiando el código de estado HTTP del punto final del Actuador que devuelve esta información, es decir, */ info.* Si nuestra aplicación resultó ser una *INSTANTÁNEA* . Obtendríamos un código de estado *HTTP* diferente .

Podemos extender fácilmente el comportamiento de un punto final predefinido utilizando las anotaciones *@EndpointExtension* , o sus especializaciones más concretas *@EndpointWebExtension* o *@EndpointJmxExtension:*

```
@Component
@EndpointWebExtension(endpoint = InfoEndpoint.class)
public class InfoWebEndpointExtension {

    private InfoEndpoint delegate;

    // standard constructor

    @ReadOperation
    public WebEndpointResponse<Map> info() {
        Map<String, Object> info = this.delegate.info();
        Integer status = getStatus(info);
        return new WebEndpointResponse<>(info, status);
    }

    private Integer getStatus(Map<String, Object> info) {
        // return 5xx if this is a snapshot
        return 200;
    }
}
```

4.9. Habilitar todos los puntos finales

Para acceder a los puntos finales del actuador utilizando HTTP, debemos habilitarlos y exponerlos . Por defecto, todos los puntos finales pero */ apagado* están habilitados. Solo los puntos finales */ health* e */ info* están expuestos de forma predeterminada.

Necesitamos agregar la siguiente configuración para exponer todos los puntos finales:

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

Para habilitar explícitamente un punto final específico (por ejemplo */ apagado)*, utilizamos:

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

Para exponer todos los puntos finales habilitados excepto uno (por ejemplo */ registradores*), utilizamos:

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

5. Resumen

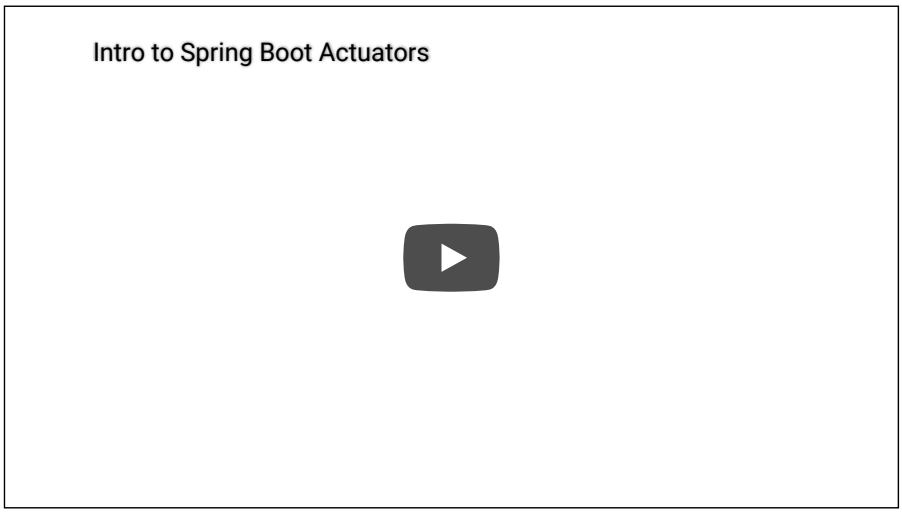
En este artículo, hablamos sobre Spring Boot Actuator. Comenzamos a definir qué significa Actuador y qué hace por nosotros.

A continuación, nos centramos en Actuator para la versión actual de Spring Boot, 1.x. discutiendo cómo usarlo, ajustarlo y extenderlo.

Luego, hablamos sobre Actuator en Spring Boot 2. Nos centramos en las novedades y aprovechamos WebFlux para exponer nuestro punto final.

Además, hablamos sobre los importantes cambios de seguridad que podemos encontrar en esta nueva iteración. Discutimos algunos puntos finales populares y cómo han cambiado también.

Por último, demostramos cómo personalizar y extender Actuator.



Como siempre podemos encontrar el código utilizado en este artículo sobre el GitHub para ambos Primavera 1.x de arranque (https://github.com/eugenp/tutorials/tree/master/spring-boot) y 2.x arranque primavera (https://github.com/eugenp/tutorials/tree/master/spring-5-reactive-security) .

▲ el más nuevo ▲ más antiguo ▲ más votado

Huésped

huésped

¿Cómo se invoca el punto final listEndpoint para ver la lista de todos los puntos finales disponibles?

✚ 0 ━ Hace 4 años

Huésped

Eugen Paraschiv (https://www.baeldung.com/)

La forma de acceder es a través de su ID personalizado, por lo que */ customEndpoint* . También actualicé el artículo. Saludos,
Eugen.

✚ 0 ━ Hace 4 años

Huésped

Rob Mitchell

¡Buen articulo! Jugué con algunos actuadores al principio cuando se lanzó el arranque de primavera y me sorprendió gratamente cuando vi un montón de URL de "salud" que se registraban en la consola Eclipse. Acabo de empezar a llamarlos para ver qué devolverían y, sí, te dan información de salud sobre tu aplicación. Con un poco de personalización, quizás podrían reemplazar algunas de las herramientas de monitoreo de producción internas más caras que existen. Gracias por el artículo. No puedo esperar para los siguientes.

✚ 0 ━

Hace 4 años

Huésped

Eugen Paraschiv (https://www.baeldung.com/)

Sí, también encontré estos puntos finales bastante útiles. También me gusta su extensibilidad: aquí hay un artículo de seguimiento que cubre métricas más avanzadas (https://www.baeldung.com/spring-rest-api-metrics) . Saludos y mantente en contacto,
Eugen.

+ 2 −

Hace 4 años

Huésped

3C273

¿Cómo se prueba?

+ 0 −

hace 3 años

Huésped

Eugen Paraschiv (https://www.baeldung.com/)

¿Cómo se prueban los actuadores? Esa es una pregunta interesante.
Primero, solo pruebe las cosas que le interesan específicamente: probar el marco en sí no es el punto aquí (ya tiene un conjunto sólido de pruebas).
En segundo lugar, los actuadores exponen datos a través de HTTP, por lo que simplemente puede usar pruebas en vivo para interactuar con estos puntos finales y probar lo que necesite. Más específicamente: puede usar el software HttpClient tranquilo para construirlos.
Espero eso ayude. Saludos,
Eugen.

+ 0 −

hace 3 años

Huésped

应 卓

¡Buen articulo! Esto es muy útil para mi. si está agregando puntos finales personalizados y haciendo esto como una biblioteca (escribiendo su inicio de arranque de resorte personalizado), debe configurar la clase a "/META-INF/spring.factories".

Me gusta esto:

```
Org.springframework.boot.actuate.autoconfigure.EndpointWebMvcConfiguration =
my.awesome.springbootstarter.MyMvcEndpointNoOne,
my.awesome.springbootstarter.MyMvcEndpointNoTwo
```

+ 0 −

hace 3 años

Huésped

Siva

¡¡Muy bien hecho!! ¿Existe alguna forma de personalizarlo para verificar / ejecutar solo algunos indicadores de Salud con el punto final de Salud y todos los demás indicadores en algún otro punto final o se pueden ejecutar todos los Indicadores de Salud personalizados en un punto final?

+ 0 −

Hace 2 años

Huésped

Grzegorz Piwowarek

¿Puedes reformular la pregunta? Estoy un poco confundido ☺

+ 0 −

Hace 2 años

Huésped

Siva

Lo siento por eso. Actualmente, el punto final de salud se configura automáticamente para varios indicadores de salud como DiskSpaceHealthIndicator, DataSourceHealthIndicator, JmsHealthIndicator, etc. Si solo quisiera verificar algunos de estos indicadores. ¿Es posible?

+ 0 −

Hace 2 años

Huésped

Grzegorz Piwowarek

Si seguro. Puede activar / desactivar HealthChecks específicos mediante las propiedades.
Por ejemplo, "management.health.mongo.enabled = false". Más información podrá encontrar en la documentación oficial. Tampoco lo sé de memoria ☺

+ 0 −

Hace 2 años

Cargar más comentarios

¡Los comentarios están cerrados en este artículo!

CATEGORÍAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)