(http://baeldung.com)

Sp Actuator

Last mo

Download
The
E-
book

Building a REST API with Spring 4?

Email Address

Download

by J ww.baeldung.com/author/jose-valero/)

Sprin gory/spring/) +

Spring /spring-boot/)

I just g 5 modules in REST With Spring:

>> CHECK OUT THE COURSE (/rest-with-spring-course#new-modules)

1. Overview

In this article, we're going to introduce Spring Boot Actuator. **We'll cover the basics first, then discuss in detail what's available in Spring Boot 1.x vs 2.x.**

We'll learn how to use, configure and extend this monitoring tool in Spring Boot 1.x. Then, we'll discuss how to do the same using Boot 2.x and WebFlux taking advantage of the reactive programming model.

Spring Boot Actuator is available since April 2014, together with the first Spring Boot release. With the upcoming release of Spring Boot 2 (http://www.baeldung.com/new-spring-boot-2), Actuator has been redesigned, and new exciting endpoints were added.

This guide is split into 3 main sections:

- **What is an Actuator?**
- **Spring Boot 1.x Actuator**

- **Spring Boot 2.x Actuator**

Further reading:

Configure a Spring Boot Web Application (http://www.baeldung.com/spring-boot-application-configuration)

Some of the more useful configs for a Spring Boot application.

Read more (http://www.baeldung.com/spring-boot-application-configuration) →

Creating a Custom Starter with Spring Boot (http://www.baeldung.com/spring-boot-custom-starter)

A quick and practical guide to creating custom Spring Boot starters.

Read more (http://www.baeldung.com/spring-boot-custom-starter) →

Testing in Spring Boot (http://www.baeldung.com/spring-boot-testing)

Learn about how the Spring Boot supports testing, to write unit tests efficiently.

Read more (http://www.baeldung.com/spring-boot-testing) →

2. What is an Actuator?

In essence, Actuator brings production-ready features to our application.

Monitoring our app, gathering metrics, understanding traffic or the state of our database becomes trivial with this dependency.

The main benefit of this library is that we can get production grade tools without having to actually implement these features ourselves.

Actuator is mainly used to **expose operational information about the running application** – health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

Once this dependency is on the classpath several endpoints are available for us out of the box. As with most Spring modules, we can easily configure or extend it in many ways.

2.1. Getting started

To enable Spring Boot Actuator we'll just need to add the *spring-boot-actuator* dependency to our package manager. In Maven:

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

Note that this remains valid regardless of the Boot version, as versions are specified in Spring Boot Bill of Materials (BOM).

3. Spring Boot 1.x Actuator

In 1.x Actuator follows a R/W model, that means we can either read from it or write to it. E.g. we can retrieve metrics or the health of our application. Alternatively, we could gracefully terminate our app or change our logging configuration.

In order to get it working, Actuator requires Spring MVC to expose its endpoints through HTTP. No other technology is supported.

3.1. Endpoints

In 1.x, Actuator brings its own security model. It takes advantage of Spring Security constructs, but needs to be configured independently from the rest of the application.

Also, most endpoints are sensitive – meaning they're not fully public, or in other words, most information will be omitted – while a handful is not e.g. */info*.

Here are some of the most common endpoints Boot provides out of the box:

- */health* – Shows application health information (a simple *'status'* when accessed over an unauthenticated connection or full message details when authenticated); it's not sensitive by default
- */info* – Displays arbitrary application info; not sensitive by default
- */metrics* – Shows 'metrics' information for the current application; it's also sensitive by default
- */trace* – Displays trace information (by default the last few HTTP requests)

We can find the full list of existing endpoints over on the official docs (http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints).

3.2. Configuring Existing Endpoints

Each endpoint can be customized with properties using the following format: *endpoints.[endpoint name].[property to customize]*

Three properties are available:

- *id* – by which this endpoint will be accessed over HTTP
- *enabled* – if true then it can be accessed otherwise not
- *sensitive* – if true then need the authorization to show crucial information over HTTP

For example, add the following properties will customize the */beans* endpoint:

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

3.3. */health* Endpoint

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

By default only health information is shown to unauthorized access over HTTP:

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

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

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

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

```
@Component
public class HealthCheck implements HealthIndicator {

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

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

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:

```java
@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;
    }
}
```

In order to access this new endpoint, its *id* is used to map it, i.e. we could exercise it hitting */customEndpoint*.

Output:

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

3.7. Further Customization

For security purposes, we might choose to expose the actuator endpoints over a non-standard port – the *management.port* property can easily be used to configure that.

Also, as we already mentioned, in 1.x. Actuator configures its own security model, based on Spring Security but independent from the rest of the application.
Hence, we can change the *management.address* property to restrict where the endpoints can be accessed from over the network:

```
#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
```

Besides, all the built-in endpoints except */info* are sensitive by default. If the application is using Spring Security – we can secure these endpoints by defining the default security properties – username, password, and role – in the application.properties file:

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

4. Spring Boot 2.x Actuator

In 2.x Actuator keeps its fundamental intent, but simplifies its model, extends its capabilities and incorporate better defaults.

Firstly, this version becomes technology agnostic. Also, it simplifies its security model by merging it with the application one.

Lastly, among the various changes, it's important to keep in mind that some of them are breaking. This includes HTTP request/responses as well as Java APIs.

Furthermore, the latest version supports now the CRUD model, as opposed to the old RW (read/write) model.

4.1. Technology Support

With its second major version, Actuator is now technology-agnostic whereas in 1.x it was tied to MVC, therefore to the Servlet API.

In 2.x Actuator defines its model, pluggable and extensible without relying on MVC for this.

Hence, with this new model, we're able to take advantage of MVC as well as WebFlux as an underlying web technology.

Moreover, forthcoming technologies could be added by implementing the right adapters.

Lastly, JMX remains supported to expose endpoints without any additional code.

4.2. Important Changes

Unlike in previous versions, **Actuator comes with most endpoints disabled**.

Thus, the only two available by default are */health* and */info*.

Would we want to enable all of them, we could set *management.endpoints.web.exposure.include=*.* Alternatively, we could list endpoints which should be enabled.

Actuator now shares the security config with the regular App security rules. Hence, the security model is dramatically simplified.

Therefore, to tweak Actuator security rules, we could just add an entry for */actuator/**:*

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

We can find further details on the brand new Actuator official docs (https://docs.spring.io/spring-boot/docs/2.0.x/actuator-api/html/).

Also, **by default, all Actuator endpoints are now placed under the */actuator* path*.***

Same as in the previous version, we can tweak this path, using the new property *management.endpoints.web.base-path.*

4.3. Predefined Endpoints

Let's have a look at some available endpoints, most of them were available in 1.x already.

Nonetheless, **some endpoints have been added, some removed and some have been restructured:**

- */auditevents* – lists security audit-related events such as user login/logout. Also, we can filter by principal or type among others fields
- */beans* – *r*eturns all available beans in our *BeanFactory*. Unlike */auditevents*, it doesn't support filtering

- */conditions* – formerly known as */autoconfig*, builds a report of conditions around auto-configuration
- */configprops* – allows us to fetch all *@ConfigurationProperties* beans
- */env* – returns the current environment properties. Additionally, we can retrieve single properties
- */flyway* – provides details about our Flyway database migrations
- */health* – summarises the health status of our application
- */heapdump* – builds and returns a heap dump from the JVM used by our application
- */info* – returns general information. It might be custom data, build information or details about the latest commit
- */liquibase* – *b*ehaves like */flyway* but for Liquibase
- */logfile* – returns ordinary application logs
- */loggers* – enables us to query and modify the logging level of our application
- */metrics* – details metrics of our application. This might include generic metrics as well as custom ones
- */prometheus* – returns metrics like the previous one, but formatted to work with a Prometheus server
- */scheduledtasks* – provides details about every scheduled task within our application
- */sessions* – lists HTTP sessions given we are using Spring Session
- */shutdown* – performs a graceful shutdown of the application
- */threaddump* – dumps the thread information of the underlying JVM

4.4. Health Indicators

Just like in the previous version, we can add custom indicators easily. Opposite to other APIs, the abstractions for creating custom health endpoints remain unchanged. However, **a new interface *ReactiveHealthIndicator* has been added to implement reactive health checks**.

Let's have a look at a simple custom reactive health check:

```
@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());
    }
}
```

A handy feature of health indicators is that we can aggregate them as part of a hierarchy. Hence, following the previous example, we could group all downstream services under a *downstream-services* category. This category would be healthy as long as every nested *service* was reachable.

Composite health checks are present in 1.x through *CompositeHealthIndicator.* Also, in 2.x we could use *CompositeReactiveHealthIndicator* for its reactive counterpart.

Unlike in Spring Boot 1.x, the *endpoints.*<id>*.sensitive* flag has been removed. To hide the complete health report, we can take advantage of the new *management.endpoint.health.show-details.* This flag is false by default.

4.5. Metrics in Spring Boot 2

In Spring Boot 2.0, the in-house metrics were replaced with Micrometer support. Thus, we can expect breaking changes. If our application was using metric services such as *GaugeService or CounterService* they will no longer be available.

Instead, we're expected to interact with Micrometer (http://www.baeldung.com/micrometer) directly. In Spring Boot 2.0, we'll get a bean of type *MeterRegistry* autoconfigured for us.

Furthermore, Micrometer is now part of Actuator's dependencies. Hence, we should be good to go as long as the Actuator dependency is in the classpath.

Moreover, we'll get a completely new response from the */metrics* endpoint*:*

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

As we can observe in the previous example, there are no actual metrics as we got in 1.x.

To get the actual value of a specific metric, we can now navigate to the desired metric, i.e., */actuator/metrics/jvm.gc.pause* and get a detailed response:

```
{
  "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"
      ]
    }
  ]
}
```

As we can see, metrics now are much more thorough. Including not only different values but also some associated meta-data.

4.6. Customizing the */info* Endpoint

The */info* endpoint remains unchanged. **As before, we can add git details using the Maven or Gradle respective dependency**:

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

Likewise, **we could also include build information including name, group, and version using the Maven or Gradle plugin:**

```xml
<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. Creating a Custom Endpoint

As we pointed out previously, we can create custom endpoints. However, Spring Boot 2 has redesigned the way to achieve this to support the new technology-agnostic paradigm.

Let's create an Actuator endpoint to query, enable and disable feature flags in our application:

```java
@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
    }

}
```

To get the endpoint, we need a bean. In our example, we're using *@Component* for this. Also, we need to decorate this bean with *@Endpoint*.

The path of our endpoint is determined by the *id* parameter of *@Endpoint*, in our case, it'll route requests to */actuator/features.*

Once ready, we can start defining operations using:

- *@ReadOperation* – it'll map to HTTP *GET*
- *@WriteOperation* – it'll map to HTTP *POST*
- *@DeleteOperation* – it'll map to HTTP *DELETE*

When we run the application with the previous endpoint in our application, Spring Boot will register it.

A quick way to verify this would be checking the logs:

```
[...].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]}"[...]
```

In the previous logs, we can see how WebFlux is exposing our new endpoint. Would we switch to MVC, It'll simply delegate on that technology without having to change any code.

Also, we have a few important considerations to keep in mind with this new approach:

- There are no dependencies with MVC
- All the metadata present as methods before (*sensitive, enabled…)* no longer exists. We can, however, enable or disable the endpoint using *@Endpoint(id = "features", enableByDefault = false)*
- Unlike in 1.x, there is no need to extend a given interface anymore
- In contrast with the old Read/Write model, now we can define *DELETE* operations using *@DeleteOperation*

4.8. Extending Existing Endpoints

Let's imagine we want to make sure the production instance of our application is never a *SNAPSHOT* version. We decided to do this by changing the HTTP status code of the Actuator endpoint that returns this information, i.e., */info.* If our app happened to be a *SNAPSHOT*. We would get a different *HTTP* status code.

We can easily extend the behavior of a predefined endpoint using the *@EndpointExtension* annotations, or its more concrete specializations *@EndpointWebExtension* or *@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;
    }
}
```

5. Summary

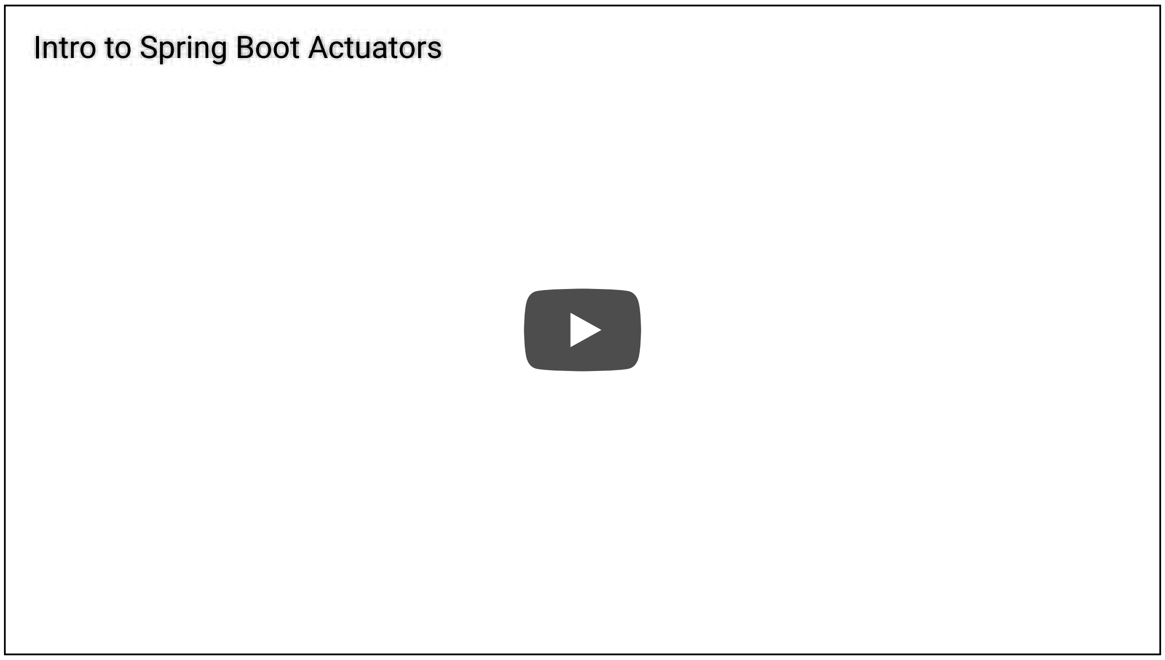
In this article, we talked about Spring Boot Actuator. We started defining what Actuator means and what it does for us.

Next, we focused on Actuator for the current Spring Boot version, 1.x. discussing how to use it, tweak it an extend it.

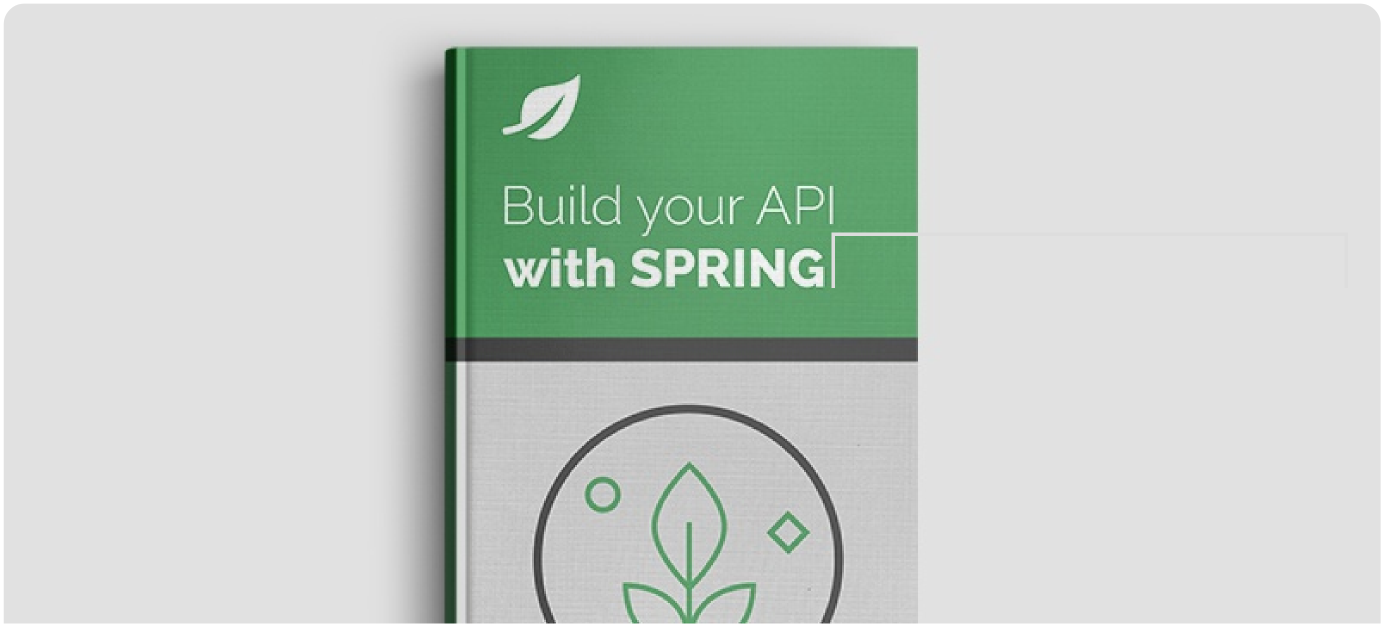
Then, we discussed Actuator in Spring Boot 2. We focused on what's new, and we took advantage of WebFlux to expose our endpoint.

Also, we talked about the important security changes that we can find in this new iteration. We discussed some popular endpoints and how they have changed as well.

Lastly, we demonstrated how to customize and extend Actuator.



As always we can find the code used in this article over on GitHub for both Spring Boot 1.x (https://github.com/eugenp/tutorials/tree/master/spring-boot) and Spring Boot 2.x (https://github.com/eugenp/tutorials/tree/master/spring-5-reactive).



(http://www.baeldung.com/wp-content/uploads/2018/04/baeldung-rest-post-footer-main-1.2.0-newcover-1.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to build your API
with Spring?

Enter your Email Address

>> Get the eBook

Subscribe

newest oldest most voted

guest

How does one invoke the listEndpoint endpoint to see the list of all available endpoints?

Guest

+ 0 − 2 years ago

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

Guest

The way to access it is via its custom id, so */customEndpoint*. I updated the article as well. Cheers, Eugen.

+ 0 − 2 years ago

Rob Mitchell

Guest

Nice article! I played around with a few actuators early on when spring boot was released and was pleasantly surprised when I saw a bunch of "health" urls logging in the Eclipse console. I just started calling them to see what they'd return and, yes, they do give you some health info about your app. With a little customization, perhaps they could replace some of the more expensive, internal production monitoring tools that are out there. Thanks for the article. Can't wait for the follow-ups.

+ 0 − 2 years ago

Guest

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

Yeah, I found these endpoints quite useful as well. I also like their extensibility – here's a followup article covering more advanced metrics (http://www.baeldung.com/spring-rest-api-metrics).
Cheers and keep in touch,
Eugen.

+ 2 − 2 years ago

Guest

3C273

How do you test?

+ 0 − 2 years ago

Guest

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

How do you test the actuators? That's an interesting question.
First – only test the things you're specifically interested in – testing the framework itself is not the point here (it already has a solid suite of tests).
Second – the actuators expose data over HTTP – so you can simply use live tests to interact with these endpoints and test whatever you need. More specifically – you can use rest-assured or HttpClient to build these out.
Hope it helps. Cheers,
Eugen.

+ 0 − 2 years ago

Guest

应卓 (https://yingzhuo.github.com)

Nice article! This is very useful to me. if you are adding custom endpoints and doing this as a library (writing your custom spring-boot-starter), you have to configuration class to "/META-INF/spring.factories".

like this:

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

+ 0 − 1 year ago

Guest

Siva

Very Nicely Done!! .Is there a way we can customize to check/run only some Health indicators with Health endpoint and all the other indicators on some other endpoint or may be run all the custom defined Health Indicators on one endpoint?

+ 0 − 1 year ago

Grzegorz Piwowarek

Guest

Can you rephrase the question? I am slightly confused 🙂

+ 0 −

1 year ago

Siva

Guest

Sorry about that. Currently the health endpoint is auto-configured for various health indicators like DiskSpaceHealthIndicator,DataSourceHealthIndicator,JmsHealthIndicator , etc. If i wanted to check only some of these indicators. is it possible?

+ 0 −

1 year ago

Grzegorz Piwowarek

Guest

Yes, sure. You can turn off/on specific HealthChecks using properties.
For example, "management.health.mongo.enabled=false". More info you will be able to find in the official documentation. I also do not know it by heart 🙂

+ 0 −

1 year ago

Load More Comments

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
#8898 (NO TITLE) (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORS (HTTP://WWW.BAELDUNG.COM/EDITORS)
MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)