Spring Boot Actuator

Raúl Estrada

Octubre 2020

1. Overview

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

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

Spring Boot Actuator is available since April 2014, together with the first Spring Boot release.

With the release of Spring Boot 2, Actuator has been redesigned, and new exciting endpoints were added.

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 become 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 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 the Spring Boot Bill of Materials (BOM).

3. Spring Boot 2.x Actuator

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

First, this version becomes technology-agnostic. It also simplifies its security model by merging it with the application one.

Among the various changes, it's important to keep in mind that some of them are breaking. This includes HTTP requests and responses as well as Java APIs.

Lastly, the latest version now supports the CRUD model as opposed to the old read/write model.

3.1. 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 as 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.

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

3.2. Important Changes

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

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

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

Actuator now shares the security config with the regular App security rules, so 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.

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*.

3.3. Predefined Endpoints

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

Also, **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 other fields.
- */beans* returns all available beans in our *BeanFactory*. Unlike */auditevents*, it doesn't support filtering.
- */conditions*, formerly known as */autoconfig*, builds a report of conditions around autoconfiguration.
- */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* summarizes 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* behaves 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.

3.4. Hypermedia for Actuator Endpoints

Spring Boot adds a discovery endpoint that returns links to all available actuator endpoints. This will facilitate discovering actuator endpoints and their corresponding URLs.

By default, this discovery endpoint is accessible through the *`/actuator`* endpoint.

Therefore, if we send a *GET* request to this URL, it'll return the actuator links for the various endpoints:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "features-arg0": {
      "href": "http://localhost:8080/actuator/features/{arg0}",
      "templated": true
    },
    "features": {
      "href": "http://localhost:8080/actuator/features",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8080/actuator/caches/{cache}",
      "templated": true
    },
    // truncated
}
```

As shown above, the */actuator* endpoint reports all available actuator endpoints under the *_links* field.

Moreover, if we configure a custom management base path, then we should use that base path as the discovery URL.

For instance, if we set the *management.endpoints.web.base-path* to */mgmt*, then we should send a request to the */mgmt* endpoint to see the list of links.

Quite interestingly, when the management base path is set to */*, the discovery endpoint is disabled to prevent the possibility of a clash with other mappings.

3.5. 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.

So, 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.

Check out our article on health indicators for a more in-depth look.

3.6. Health Groups

As of Spring Boot 2.2, we can organize health indicators into groups and apply the same configuration to all the group members.

For example, we can create a health group named *custom* by adding this to our *application.properties*:

```
management.endpoint.health.group.custom.include=diskSpace,ping
```

This way, the *custom* group contains the *diskSpace* and *ping* health indicators.

Now if we call the */actuator/health* endpoint, it would tell us about the new health group in the JSON response:

```
{"status":"UP","groups":["custom"]}
```

With health groups, we can see the aggregated results of a few health indicators.

In this case, if we send a request to */actuator/health/custom*, then:

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

We can configure the group to show more details via *application.properties*:

```
management.endpoint.health.group.custom.show-components=always
management.endpoint.health.group.custom.show-details=always
```

Now if we send the same request to */actuator/health/custom*, we'll see more details:

```
{
  "status": "UP",
  "components": {
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 499963170816,
        "free": 91300069376,
        "threshold": 10485760
      }
    },
    "ping": {
      "status": "UP"
    }
  }
}
```

It's also possible to show these details only for authorized users:

```
management.endpoint.health.group.custom.show-components=when_authorized
management.endpoint.health.group.custom.show-details=when_authorized
```

We can also have a custom status mapping.

For instance, instead of an HTTP 200 OK response, it can return a 207 status code:

```
management.endpoint.health.group.custom.status.http-mapping.up=207
```

Here, we're telling Spring Boot to return a 207 HTTP status code if the *custom* group status is *UP*.

3.7. Metrics in Spring Boot 2

In Spring Boot 2.0, the in-house metrics were replaced with Micrometer support, so 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 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, so 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 see, 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, e.g., */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"
      ]
    }
  ]
}
```

Now metrics are much more thorough, including not only different values but also some associated metadata.

3.8. Customizing the */info* Endpoint

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

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

3.9. 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:

```
@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 is to check 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. If 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 exist. 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, we can now define *DELETE* operations using *@DeleteOperation*.

3.10. Extending Existing Endpoints

Let's imagine we want to make sure the production instance of our application is never a *SNAPSHOT* version.

We decide 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;
    }
}
```

3.11. Enable All Endpoints

In order to access the actuator endpoints using HTTP, we need to both enable and expose them.

By default, all endpoints but */shutdown* are enabled. Only the */health* and */info* endpoints are exposed by default.

We need to add the following configuration to expose all endpoints:

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

To explicitly enable a specific endpoint (e.g., */shutdown),* we use:

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

To expose all enabled endpoints except one (e.g., */loggers*), we use:

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

4. Spring Boot 1.x Actuator

In 1.x, Actuator follows a read/write model, which means we can either read from it or write to it.

For example, 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.

4.1. 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 most information will be omitted — while a handful are 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; it's not sensitive by default.
- */metrics* shows metrics information for the current application; it's 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.

4.2. Configuring Existing Endpoints

We can customize each endpoint with properties using the 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, adding the following properties will customize the */beans* endpoint:

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

4.3. */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, etc.

By default, unauthorized users can only see status information when they 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, and more.

Note that this only works for Spring Boot versions below 1.5.0. For 1.5.0 and later versions, we should also disable security by setting *management.security.enabled=false* for unauthorized access.

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("myHealthCheck")
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:

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

4.4. */info* Endpoint

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

```
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"
    }
}
```

4.5. */metrics* Endpoint

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

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 (single-value snapshots of data) and counters, i.e., incrementing/decrementing metrics.

Let's implement our own custom metrics into the */metrics* endpoint.

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.

4.6. Creating a New Endpoint

In addition to using the existing endpoints provided by Spring Boot, we can also create an entirely new one.

First, we 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. In other words we could exercise it hitting */customEndpoint*.

Output:

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

4.7. 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 propertie (username, password, and role) in the *application.properties* file:

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

5. Conclusion

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

Next, we focused on Actuator for the current Spring Boot version 2.x, discussing how to use it, tweak it, and extend it. We also 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.

Then we discussed Actuator in the earlier Spring Boot 1 version.

Lastly, we demonstrated how to customize and extend Actuator.