Spring Boot Actuator and Health Indicators

Utilizing Spring Boot's built-in metrics & health indicators and adding your own

1.18.5

Objectives

After completing this lesson, you should be able to do the following

- Configure which Spring Boot Actuator HTTP endpoints are to be enabled and exposed
- Secure Spring Boot Actuator HTTP endpoints
- Define custom metrics
- Define custom health indicators

Agenda

- **Spring Boot Actuator**
- Setting up Actuator
- Metrics
- Health indicators
- External monitoring systems

Actuator

What value does it provide?

Actuator provides:

- Production grade monitoring without having to implement it yourself
- A framework to easily gather and return metrics and health indicators
- Integration with 3rd party monitoring system for aggregation and visualization

Actuator

How does it work?

The Actuator library adds many production-ready monitoring features

Accessible as HTTP endpoints:

- **`/actuator/info`**
- **`/actuator/health`**
- **`/actuator/metrics`**

More endpoints

/actuator/info

General data, custom data, build information or details about the latest commit

```
{
    "build":   {
        "version": "5.3.23",
        "artifact": "37-actuator",
        "name": "37-actuator",
        "group": "io.spring.training.core-spring",
        "time": "2022-03-25T22:06:18.311Z"
    }
}
```

/actuator/health

Application health status

- Default output is minimal

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

/actuator/metrics

List of generic and custom metrics measured by the application

- *Not* exposed by default

```
{
    "names": [
        "jvm.memory.max",
        "jvm.gc.memory.promoted",
        "http.server.requests",
        "system.cpu.usage",
        "hikaricp.connections.active",
        "process.start.time",
        "reward.summary",
        ...
    ]
}
```

Custom Metric

Example: /actuator/metrics/http.server.requests

```
{
 "name": "http.server.requests",
 "measurements": [
   { "statistic": "COUNT", "value": 13 },
   { "statistic": "MAX", "value": 0.003785154 },
   ...
 ],
 "availableTags": [ {
    "tag": "method",
    "values": [ "POST", "GET" ],
     ...
}
```

Agenda

- Spring Boot Actuator
- **Setting up Actuator**
- Metrics
- Health indicators
- External monitoring systems

Adding the Actuator dependency

Include the Spring Boot actuator starter

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

Some of the Available Actuator Endpoints - 1

`beans`	Spring Beans created by application
`conditions`	Conditions used by Auto-Configuration
`env`	Properties in the Spring Environment
`health`	Current state of the application
`configprops`	Collated list of all @ConfigurationProperties
`info`	Arbitrary application information
`loggers`	Query and modify logging levels
`mappings`	Spring MVC request mappings

Some of the Available Actuator Endpoints - 2

`metrics`	List of available metrics
`session`	Fetch or delete user sessions (only if using Spring Session)
`shutdown`	Shutdown the application (gracefully), disabled by default
`threaddump`	Performs a thread dump

For a full list see:

https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html#production-ready-endpoints-exposing-endpoints

Actuator Endpoints: Enabled vs Exposed

Enabled = given endpoint is created and its bean exists in the application context

Default = all endpoints enabled except *shutdown*

Exposed = given endpoint is accessible via JMX or HTTP

HTTP Default = only *health* exposed

JMX Default = all enabled endpoints are exposed

Note:

- JMX capability is only available when `spring.jmx.enabled=true`
- HTTP capability is only available when using Spring MVC, WebFlux or Jersey

HTTP Actuator Endpoints

Mapped to **`/actuator/xxx`** by default - customizable

```
# Change actuator base path
management.endpoints.web.base-path=/admin
```

For security reasons, only one endpoint is exposed by default

- **`/actuator/health`**
- Secure actuator URLs using Spring Security

Exposing HTTP Endpoints

If endpoints exposed explicitly, defaults overridden

```
# Default setup
management.endpoints.web.exposure.include=health
```

```
# Expose just beans, env and info endpoints
# NOTE: health and info not exposed unless listed
management.endpoints.web.exposure.include=beans,env,info
```

```
# Expose all endpoints
management.endpoints.web.exposure.include=*
```

Secure Endpoints - Aligned with Spring Security

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests((authz) -> authz
        .requestMatchers("/actuator/health").permitAll()
        .requestMatchers("/actuator/**").hasRole("ACTUATOR")
        .anyRequest().authenticated());
    return http.build();
}
```

Agenda

- Spring Boot Actuator
- Setting up Actuator
- **Metrics**
- Health indicators
- External monitoring systems

Metrics

How do you collect metrics?

- Since Spring Boot 2.0, uses Micrometer library
  - Multi-dimensional metrics
- It instruments your JVM-based application code without vendor lock-in
  - SLF4J for metrics
- Designed to add little to no overhead to your metrics collection activity

Custom metrics

What are they?

Custom metrics can be measured using Micrometer classes such as **`Counter`**, **`Gauge`**,**`Timer`**, and **`DistributionSummary`**.

- Classes are created or registered with a **`MeterRegistry`** bean
- Custom metric names are listed on the **`/actuator/metrics`** endpoint
- Custom metric data can be fetched at **`/actuator/metrics/[custom-metric-name]`**

Hierarchical vs Dimensional Metrics

- How can you access metrics data, i.e. on http requests?
- You want to use arbitrary combination of
  - Http method, URI, Response status, Exception status
  - Custom attributes
- Example metrics data on http requests
  - Http requests whose Http method is GET and Response status is 200
  - Http requests whose Http method is POST and “Region-of-origin” custom attribute is “us-east”

Hierarchical Metrics

- Often follow a naming convention that embeds key/value attribute pairs into the name separated by periods
  - http.method.<method-value>.status.<status-value>
- Examples
  - Http.method.get.status.200
  - http.method.get.status.*
- Characteristics
  - Consistent naming convention is hard to achieve
  - Adding new attribute could break existing queries

Dimensional Metrics

- Metrics are tagged (a.k.a. dimensional)
- Examples
  - http?tag=method:get&tag=status:200
  - http?tag=method:get&tag=status:200&tag=region:us-east
- Characteristics
  - Flexible naming convention
  - Adding a new attribute to a query is easy

MeterRegistry - Timer

```
public class OrderController {
    private Timer timer;

    public class OrderController(MeterRegistry registry) {
        this.timer = registry.timer("orders.submit"));
    }

    @PostMapping("/orders")
    public Order placeOrder( ... ) {
        return timer.record( () -> { /* lambda: code placing an order … */ } );
    }

    @GetMapping("/orders")
    @Timed("orders.summary")
    public List<Order> orderSummary() {...}
}
```

Can also create counters, gauges or summaries

`Timer` is part of Micrometer project

`@Timed` avoids mixing of concerns

Timer provides count, mean, max and total of its metric

Recording to a `DistributionSummary`

```
@Controller
public class RewardController {
  private final DistributionSummary summary;

  public RewardController(MeterRegistry meter) {
    summary = DistributionSummary.builder("reward.summary")
                                 .baseUnit("dollars")
                                 .register(meter);
  }

  @PostMapping(value = "/rewards")
  public ResponseEntity<Void> create(@RequestBody Reward reward) {
    summary.record(reward.amount);
    ...
  }
}
```

Build a meter and register it

Distribution Summary provides a count, total, and max value for its metric

Example: /actuator/metrics/reward.summary

```
{
   "name": "reward.summary",
   "measurements": [
        {
           "statistic": "COUNT",
           "value": 3
        },
        {
           "statistic": "TOTAL",
           "value": 13
           ...
```

Agenda

- Spring Boot Actuator
- Setting up Actuator
- Metrics
- **Health indicators**
- External monitoring systems

/actuator/health

Application health status

By default, health endpoint shows only basic health information.

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

More Health Details Possible

```
{
  "status": "UP",
  "details": {
    "db": {
      "status": "UP",
      "details": {
        "database": "MySQL",
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 499963170816,
        ...
```

application.properties

```
# Set Spring Boot property
management.endpoint.health.show-details=always
```

Default: a validation query, such as 'select 1' returns successfully

Default: < 10MB free is down

Group Health Indicators

application.properties

```
# Group health indicators
management.endpoint.health.group.<group-name>.include=<health
indicators>
```

```
management.endpoint.health.group.system.include=diskSpace,db
management.endpoint.health.group.web.include=ping
```

```
{
  "status": "UP",
  "groups": {
     "system",
     "web",
     ...
  }
}
```

Access it via
http://localhost:8080/actuator/health

Configure Health Indicator Group Individually

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "MySQL",
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 499963170816,
        ...
```

application.properties

```
# Configure "system" health indicator group
management.endpoint.health.group.system.show-details=always
```

Access it via
http://localhost:8080/actuator/health/system

List of Auto-configured HealthIndicators

- Many health-indicators setup automatically
  - Providing their dependencies are on the classpath
  - Disk Space, DataSource, Cassandra, Elasticsearch, InfluxDb, JMS, Mail, MongoDB, Neo4J, RabbitMQ, Redis, Solr, …

- Full details

  https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html#production-ready-health-indicators

/actuator/health

Custom health indicators

Custom health indicators can be added to the **`/actuator/health`** endpoint and will be rolled up into the overall application health status.

- Create a class which implements **`HealthIndicator`** interface
  - Override the **`health()`** method to return the status
- Or extend **`AbstractHealthIndicator`**
  - Override the **`doHealthCheck()`** method

/actuator/health

Health indicator statuses

- Built in status values
  - `DOWN`
  - `OUT_OF_SERVICE`
  - `UNKNOWN`
  - `UP`
- Severity order can be overridden using

```
management.endpoint.health.status.order=
FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

Implementing a custom Health Indicator

```
@Component
public class MyCustomHealthCheck implements HealthIndicator {

  @Override
  public Health health() {

      if (!customHealthValidationCheck()) {
          return Health.down().build();
      } else {
          …
```

/actuator/health

```
{
  "status": "DOWN",
  "details": {
     "myCustomHealthCheck": {
          "status": "DOWN"
     },
     "db": {
        "status": "UP",
        ...
     },
     "diskSpace": {
        "status": "UP",
        ...
```

Custom Health Indicator

Adding Detailed Health Indicator Information

```
@Component
public class MyCustomHealthCheck implements HealthIndicator {

   @Override
   public Health health() {

       if (!customHealthValidationCheck()) {
           return Health.down().withDetail("metricName",0).build();
       } else {
           …
```

Adding Detailed Health Indicator Information

```
@Component
public class MyCustomHealthCh

  @Override
  public Health health() {

     if (!customHealthValidationCh
        return Health.down().with
     } else {
        …
```

```
{
  "status": "DOWN",
  "details": {
    "myCustomHealthCheck": {
        "status": "DOWN",
        "details": {
          "metricName": 0
        }
    },
    "db": {
       "status": "UP",
       …
```

Agenda

- Spring Boot Actuator
- Setting up Actuator
- Metrics
- Health indicators
- **External monitoring systems**

Actuator data

What do we do with the data?

- Actuator alone doesn’t provide anything except REST endpoints
- To truly add value, this data needs to be gathered, persisted, aggregated, and visualized for easy consumption

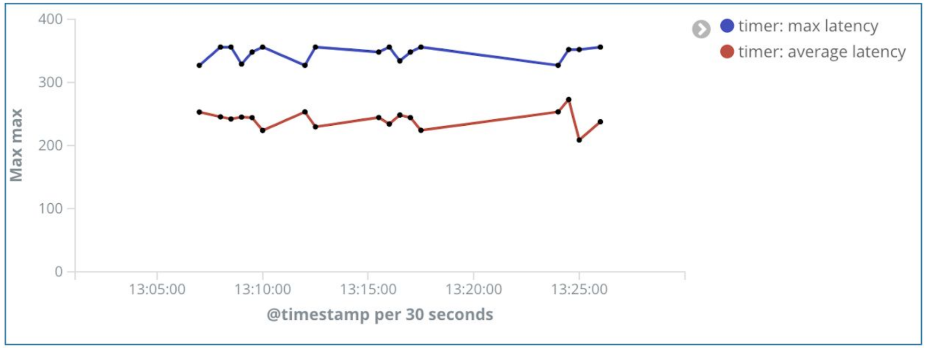
Integration Options

External monitoring systems that can be integrated with Actuator

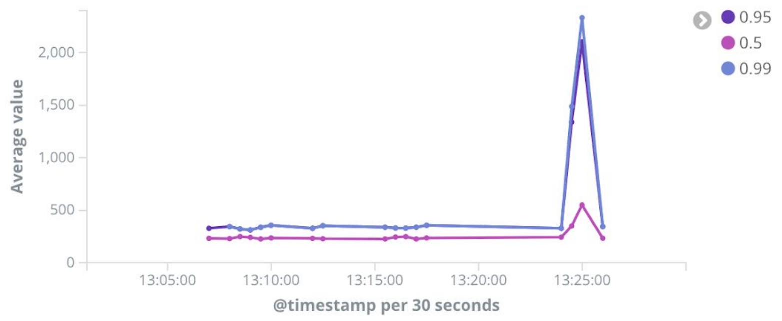
- Atlas (Netflix)
- CloudWatch
- Datadog
- Dynatrace
- Ganglia
- Graphite
- InfluxDB
- JMX
- New Relic
- Prometheus
- SignalFx
- StatsD
- Wavefront (VMware)

Monitoring with timer

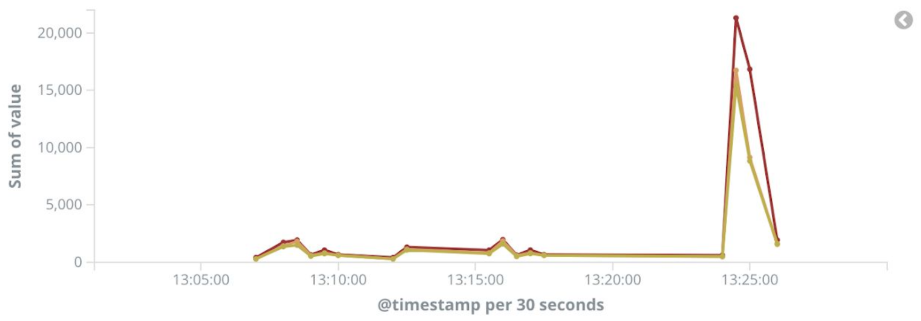
Timer latency



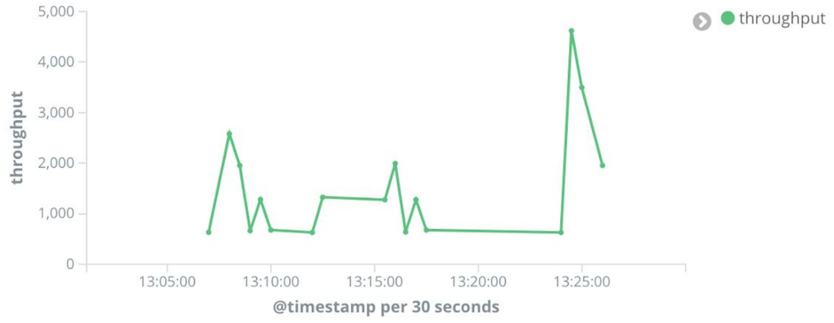
Percentiles



SLAs



Throughput



More info on external monitoring

https://spring.io/blog/2018/03/16/micrometer-spring-boot-2-s-new-application-metrics-collector

Summary

What do you remember?

- What are benefits of Actuator?
- What is the default exposed endpoint?
- What does the health indicator endpoint tell you about your application?

Lab: Play with healthchecks and metrics

https://github.com/Nimedas/imt-spring-2025

Branche : 4-solution