



Pivotal®

Spring Boot Actuator and Health Indicators

Utilizing Spring Boot's Built-in metrics
and adding your own



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
- Integration with 3rd party dashboards to show health and metrics of distributed systems

Actuator

How does it work?

The Actuator library adds many production-ready monitoring features

Accessible via JMX

Or as HTTP endpoints:

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

[More endpoints](#)

Important Note

Content applies to *Spring Boot 2.x*

Actuator implementation was very different in Spring Boot 1.x

Objectives

After completing this lesson, you should be able to

- Configure which Spring Boot Actuator HTTP endpoints are to be exposed
- Secure Spring Boot Actuator HTTP endpoints
- Define custom metrics
- Extend health endpoint to add custom health checks

Agenda

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



/actuator/info

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

```
{
  "build": {
    "version": "5.0.c.RELEASE",
    "artifact": "37-actuator",
    "name": "37-actuator",
    "group": "io.pivotal.education.core-spring",
    "time": "2018-07-31T22:06:18.311Z"
  }
}
```

/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
httptrace	Most recent HTTP requests in Web application
info	Displays 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
jolokia	Exposes JMX beans over HTTP (not just actuators)

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 are enabled except *shutdown*

httptrace, auditevents (from Spring Boot 2.2.0)

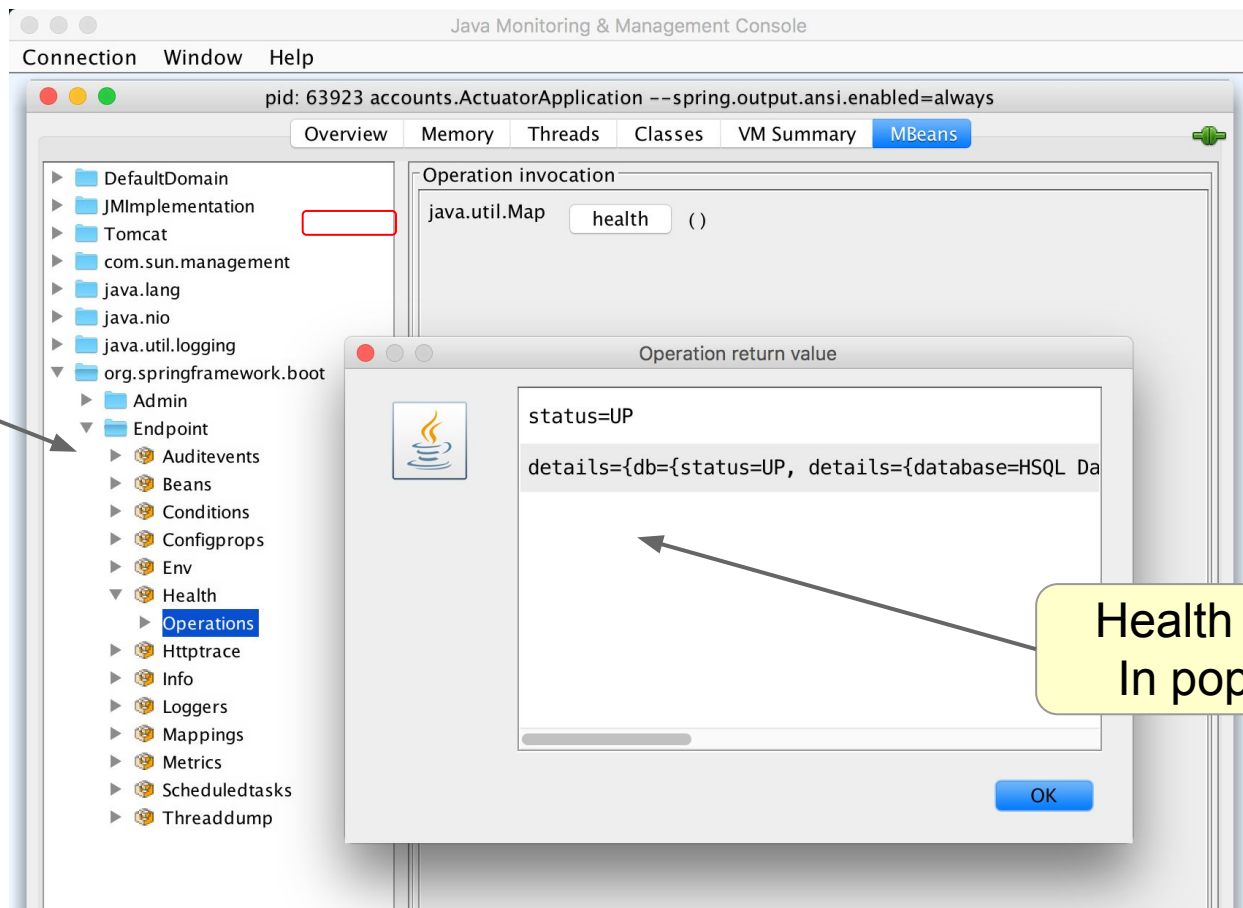
Exposed = given endpoint is accessible via JMX or HTTP

JMX Default = All exposed

HTTP Default = only *info* and *health* exposed

JMX Endpoints - Exposed By Default

Endpoints in
JConsole



Health Info
In popup

Controlling Exposure of JMX Endpoints

- The exposure of JMX endpoints can be controlled using properties
- Either property can take a comma-separated list of actuators to include/exclude

```
# Default setup: all exposed, no exclusions  
management.endpoints.jmx.exposure.exclude=  
management.endpoints.jmx.exposure.include=*
```

HTTP Actuator Endpoints

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

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

- For security reasons, only two endpoints are exposed by default
 - `/actuator/health`
 - `/actuator/info`
 - Secure actuator URLs using Spring Security

Exposing HTTP Endpoints

- If endpoints exposed explicitly, defaults overridden

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

```
# 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

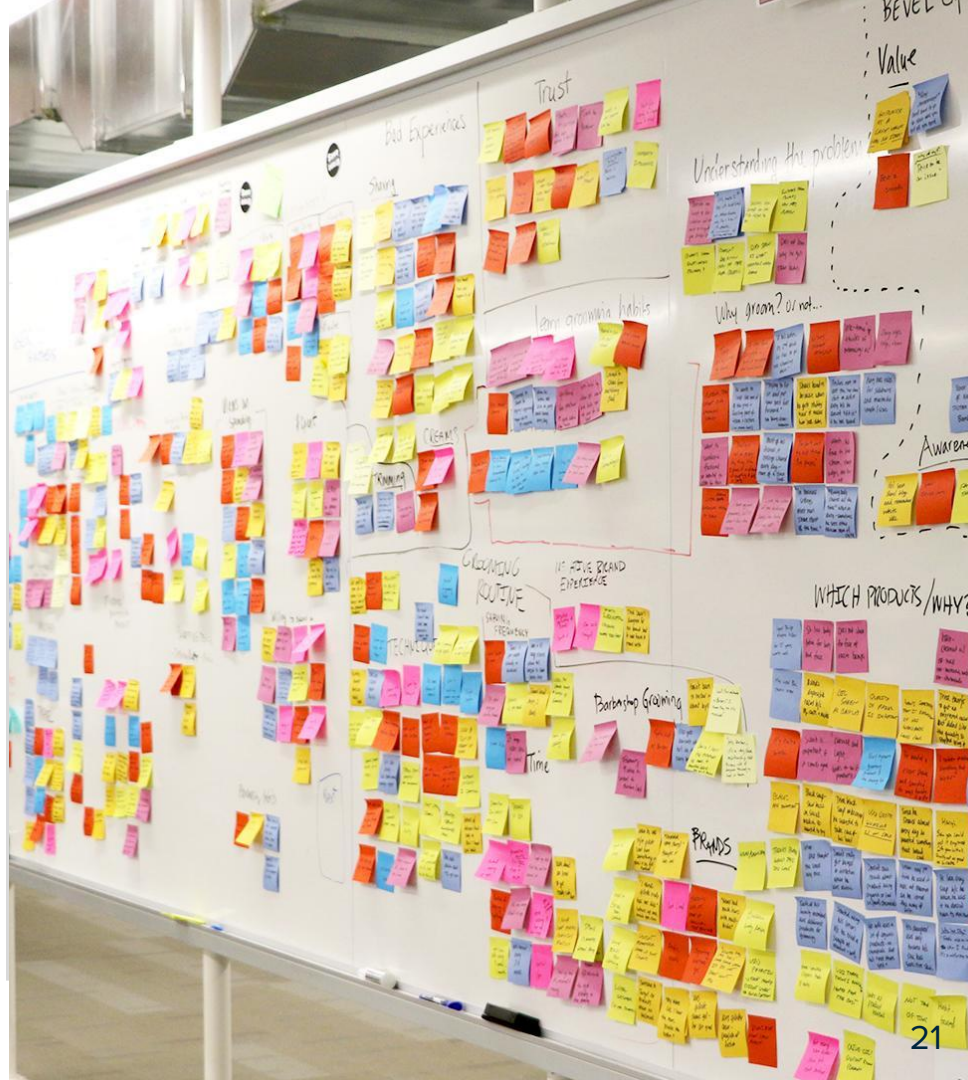
```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .mvcMatchers("/actuator/health", "/actuator/info").permitAll()  
        .mvcMatchers("/actuator/**").hasRole("ACTUATOR")  
        .anyRequest().authenticated();  
}
```

// Base-path agnostic

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .requestMatchers(EndpointRequest.to(HealthEndpoint.class, InfoEndpoint.class)).permitAll()  
        .requestMatchers(EndpointRequest.toAnyEndpoint()).hasRole("ACTUATOR")  
        .anyRequest().authenticated();  
}
```

Agenda

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



Metrics

How do you collect metrics?

- 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 (using Spring Boot 2.0 actuator)
 - `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("bytes")  
            .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

application.properties

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

```
# 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

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-endpoints.html#_auto_configured_healthindicators

/actuator/health

Custom health checks

Custom health checks 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 Check

```
@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 Metric

Adding Detailed Health Check Information

```
@Component
public class MyCustomHealthCheck implements HealthIndicator {

    @Override
    public Health health() {

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

Adding Detailed Health Check Information

```
@Component
public class MyCustomHealthCheck {

    @Override
    public Health health() {

        if (!customHealthValidationCheck()) {
            return Health.down().withDetail("customHealthCheck", "DOWN");
        } 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, aggregated, and graphed for easy consumption

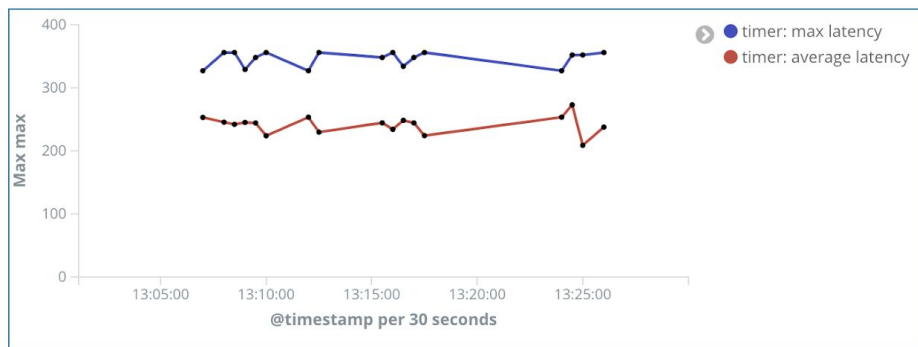
Integration options

External monitoring systems that can be integrated with Actuator

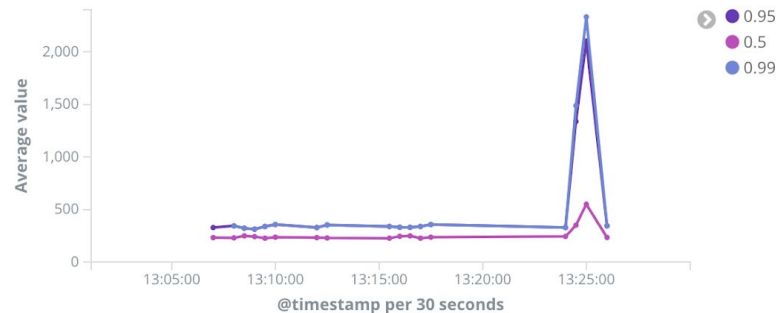
- Netflix Atlas
- CloudWatch
- Datadog
- Dynatrace
- Ganglia
- Graphite
- InfluxDB
- JMX
- New Relic
- Prometheus
- SignalFx
- StatsD
 - Etsy, dogstatsd, Telegraf
- Wavefront

Monitoring with timer

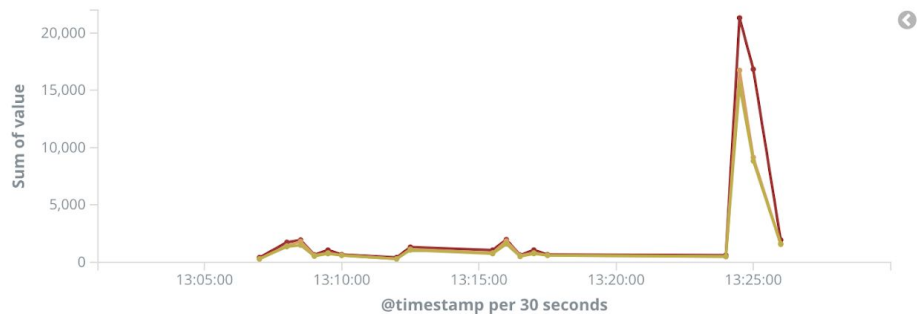
Timer latency



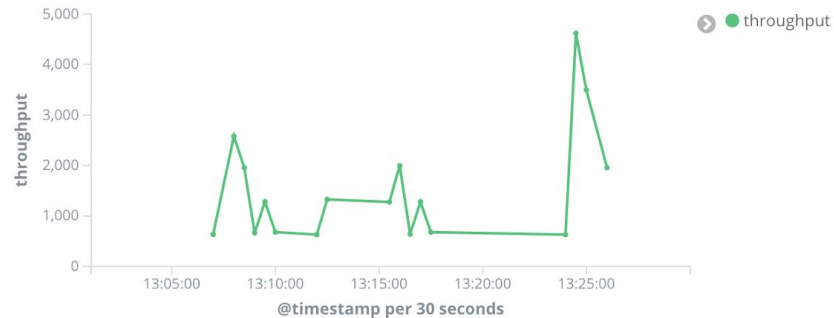
Percentiles



SLAs



Throughput



More info on external monitoring

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

Agenda


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



Summary

What do you remember?

- What are three benefits of Actuator?
- What are the two default endpoints exposed?
- What does the health check endpoint tell you about your application?

A man and a woman are sitting at a desk in a lab, looking at a computer monitor. The man is on the left, and the woman is on the right. They are both looking at the screen with interest. The background is slightly blurred, showing other people and equipment in the lab.

***Lab:* Enabling standard actuators, adding custom metrics and custom health indicators**

**Lab project:
44-actuator**

**Anticipated Lab time:
30 Minutes**