Spring Boot Observability in Practice

Actuator, Micrometer, and OpenTelemetry

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com

Abstract

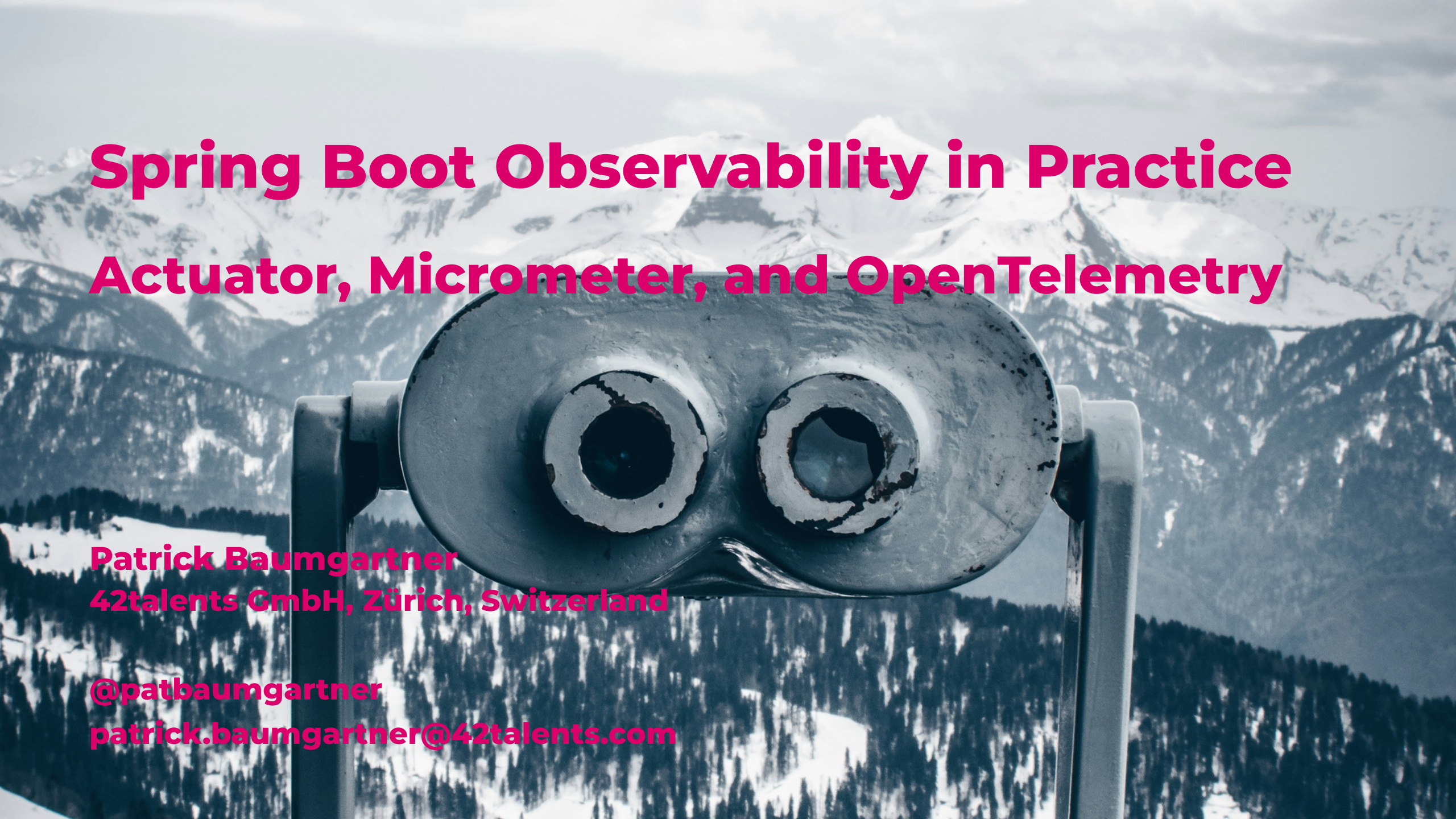
Spring Boot Observability in Practice: Actuator, Micrometer, and OpenTelemetry

We've all been there: something goes wrong in production, and the logs don't tell the whole story. In this talk, we go beyond basic health checks and show how Spring Boot Actuator and Micrometer help you build applications that are not just functional, but also transparent, diagnosable, and production-ready.

Through practical examples, we enable and customize Actuator endpoints to gain real-time insights into your application's state. We collect metrics with Micrometer and export them to Prometheus. We also visualize metrics and set up alerts in Grafana.

Another focus is integrating OpenTelemetry for distributed tracing across multiple services. We close with best practices for embedding observability into your development process in a sustainable way.

Whether you work locally or in Kubernetes, this session shows how to improve the observability of your Spring Boot application without getting lost in technical complexity.



Spring Boot Observability in Practice

Actuator, Micrometer, and OpenTelemetry

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com

Spring Boot Observability in Practice Actuator, Micrometer, and OpenTelemetry

Patrick Baumgartner / 42talents GmbH

Who Am I?

Patrick Baumgartner

Technical Agile Coach @ **42talents**

Focus on the **development of software solutions with humans**

Coaching, Architecture, Development, Reviews, Training

Lecturer @ **Zurich University of Applied Sciences ZHAW**

Co-Organizer of **Voxxed Days Zurich** and **JUG Switzerland**, Java Champion, Oracle ACE Pro Java

@patbaumgartner

Agenda

Agenda

1. From Monitoring to Observability
2. Enabling Observability in Spring Boot
3. Actuator Deep Dive and Endpoint Security
4. Metrics with Micrometer
5. Distributed Tracing with OpenTelemetry
6. Logging, Structured Logging, and Auditing
7. Best Practices on CI/CD and Kubernetes
8. SBOM, HTTP Exchanges, and Ops Hygiene
9. Wrap Up and Q&A

From Monitoring to Observability

Logs vs. System Understanding

Logs Tell You *What* Happened, Not *Why*

- Logs show discrete events and can miss full system context
- In distributed, asynchronous, and containerized systems, log context often gets lost
- To understand behavior we need visibility across **requests, systems, and time**

The Three Pillars of Observability

Logs, Metrics, Traces

- **Logs** - event based and detailed, but high in volume
- **Metrics** - numeric trends, fast to query (Micrometer + Prometheus + Grafana)
- **Traces** - request flow across services (OpenTelemetry + Micrometer Tracing)

Together these signals provide a **complete view** of system health and performance.

Observability vs. Monitoring vs. Telemetry

Understanding the Differences

- **Monitoring** - collects known metrics and alerts when thresholds break
- **Observability** - ability to ask new questions without redeploying code
- **Telemetry** - raw data that enables observability: metrics, traces, logs

Observability helps debug **unknown unknowns** and answer *why* something happened.

Spring Boot Observability Stack

Built-In Tools and Integrations

- **Actuator** - exposes runtime insights such as `/actuator/health`, `/metrics`, `/info`
- **Micrometer** - unified metrics facade for many backends
- **OpenTelemetry and Micrometer Tracing** - distributed tracing with OTLP, Zipkin, and others

Spring Boot auto-configures these components into a cohesive observability experience.

The Goal: Diagnosable and Production-Ready Apps

Build Trust Through Visibility

- Expose application state, performance, and dependencies through consistent instrumentation
- Troubleshoot fast without redeploying
- Build shared trust between **developers** and **operations** with observability by design
- Works both **locally** and in **Kubernetes** environments

Enabling Observability in Spring Boot

Foundation for Production-Ready Insights

Add Spring Boot Actuator

One Dependency to Enable the Features

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

or Gradle:

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
```

- Adds endpoints for **health**, **metrics**, **environment**, **logging**, and **info**
- Auto-configures based on classpath: Micrometer, OpenTelemetry, Prometheus, and more

Default Actuator Endpoints

What You Get Out of the Box

Accessible under `/actuator/*`:

- `/actuator/health` - application and dependency health
- `/actuator/info` - build, git, and app info
- `/actuator/metrics` - Micrometer metrics overview
- `/actuator/loggers` - inspect or change log levels
- `/actuator/env`, `/actuator/beans`, `/actuator/threaddump` - deep introspection tools

By default only `/health` is exposed over HTTP. Secure anything else you expose.

Exposing More Endpoints

Configure with YAML or Environment Variables

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,loggers
  endpoint:
    health:
      show-details: when-authorized
```

- Supports properties, YAML, and environment variables
- Example: `MANAGEMENT_ENDPOINTS_WEB_EXPOSURE_INCLUDE=*`
- Customize base path or port for management traffic

Management Port, Path, and SSL

```yaml
management:
  server:
    port: 8081
    address: 127.0.0.1
  endpoints:
    web:
      base-path: "/manage"
```

- Separate port can reduce risk and simplify routing
- Address limit such as `127.0.0.1` for local-only access
- Management SSL can differ from application SSL if needed

Environment-Specific Exposure

Tailor per Environment

- **Local** - enable all endpoints for debugging
- **Staging** - simulate production telemetry such as Prometheus scrape and OTLP export
- **Production** - restrict exposure, secure endpoints, centralize metrics and traces

```
# application-local.yaml
management.endpoints.web.exposure.include: "*"
management.endpoint.health.show-details: always

# application-prod.yaml
management.endpoints.web.exposure.include: "health,info,prometheus"
management.endpoint.health.show-details: never
management.server.port: 8081
```

Actuator Deep Dive and Endpoint Security

Powering Observability in Spring Boot

What Actuator Brings

Observability Foundation

- Monitor, inspect, and interact with running applications
- Endpoints available via **HTTP** or **JMX**
- Auto-configuration based on classpath such as Micrometer and Security

Tip: Only `/health` is exposed by default. Configure more with care.

Common Actuator Endpoints

Endpoint	Description
`/actuator/health`	Application and dependency health
`/actuator/info`	Build and git metadata
`/actuator/metrics`	Micrometer metrics overview
`/actuator/loggers`	Inspect or change log levels
`/actuator/env`	View environment and profiles
`/actuator/prometheus`	Prometheus metrics when registry present
`/actuator/startup`	Application startup phases
`/actuator/sbom`	Software Bill of Materials when configured

Endpoint Exposure and Access Control

What Is Accessible and to Whom

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,loggers
  endpoint:
    loggers:
      access: read-only
  endpoints:
    access:
      default: none
```

- Use `management.endpoints.*.exposure.*` to expose over HTTP or JMX
- Use `management.endpoint.<id>.access` to control access level such as `read-only`

Discovery, Caching, and Path Mapping

Taming the Surface and Responses

```yaml
management:
  endpoints:
    web:
      discovery:
        enabled: false
      base-path: "/manage"
      path-mapping:
        health: "healthcheck"
  endpoint:
    beans:
      cache:
        time-to-live: 10s
```

HTTP Endpoint Hygiene

When and How to Enable

- `heapdump` and `logfile` only when necessary and with access control
- `httpexchanges` for HTTP exchange history in dev and staging
- JSON actuator responses require Jackson on the classpath

Separate Management Port and CORS

Best Practices

- Secure endpoints with **Spring Security**
- Run management on another port
- Enable CORS for dashboards and internal tooling

```
management.server.port: 8081
management.server.address: 127.0.0.1
management.endpoints.web.cors.allowed-origins: "https://grafana.example.com"
management.endpoints.web.cors.allowed-methods: "GET,POST"
```

Health Checks and Kubernetes Probes

Application Health and Readiness

- `/actuator/health` - composite health such as DB, disk, Redis
- `/actuator/health/liveness` - for container liveliness
- `/actuator/health/readiness` - for startup readiness

Kubernetes example:

```
livenessProbe:
  httpGet:
    path: /actuator/health/liveness
    port: 8080
readinessProbe:
  httpGet:
    path: /actuator/health/readiness
    port: 8080
```

Health Groups and Additional Paths

Customize Probes and Targets

```
management.endpoint.health.group.db.include: db
management.endpoint.health.probes.add-additional-paths: true # exposes /livez and /readyz on main port
```

Using /info for Build

Know What Is Deployed

Maven

```
<execution>
  <goals><goal>build-info</goal></goals>
</execution>
```

Gradle

```
springBoot {
  buildInfo()
}
```

Git Metadata

Know What Is Deployed

Add Git metadata:

```
management.info.git.mode: full
```

Example output:

```
{
  "build": { "version": "1.2.0", "time": "2025-10-31T12:00Z" },
  "git": { "branch": "main", "commit": { "id": "abc123" } }
}
```

Dynamic Log Level Control

`/actuator/loggers` in Action

```
GET  /actuator/loggers
POST /actuator/loggers/com.example.service
{ "configuredLevel": "DEBUG" }
```

- On-the-fly diagnostics without restart
- Works with Logback and Log4j2

Process Monitoring Files

PID and Port Tracking for Ops

`ApplicationPidFileWriter` and `WebServerPortFileWriter` can write PID and port files for orchestration and diagnostics.

`META-INF/spring.factories` example:

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.context.ApplicationPidFileWriter,\
org.springframework.boot.web.context.WebServerPortFileWriter
```

Custom Endpoint

Example

```
@Endpoint(id = "greeting")
public class GreetingEndpoint {
  @ReadOperation
  public Map<String, String> greet() {
    return Map.of("message", "Hello from Actuator!");
  }
}
```

Custom Health Indicator

Example

```
@Component
class MyServiceHealthIndicator implements HealthIndicator {
  public Health health() {
    boolean ok = checkService();
    return ok ? Health.up().build()
              : Health.down().withDetail("error", "Timeout").build();
  }
}
```

Metrics with Micrometer

Observability Through Metrics

Micrometer in a Nutshell

The Metrics Facade

- Micrometer is the **SLF4J of metrics** - one API, many backends
- Integrated with Spring Boot Actuator - metrics under `/actuator/metrics`
- Core concept: **MeterRegistry** collects counters, gauges, timers, and more
- Spring Boot creates a **composite registry** and adds registries for each backend on the classpath

Built-In Metrics

Instant Visibility

Spring Boot auto-instruments:

- **JVM** - memory, GC, threads, classes, virtual threads when available
- **System** - CPU, disk, uptime, file descriptors
- **HTTP** - `http.server.requests` with tags such as method, status, URI
- **Database pools** - active and max
- **Caches** - hits, misses, evictions
- **Thread pools** - utilization and scheduling
- **Startup** - `application.started.time` , `application.ready.time`

All available via `/actuator/metrics` and drillable by name and tags.

Custom Metrics with MeterRegistry

Create Counters, Gauges, Timers Programmatically

```
@Component
class CheckoutService {
  private final Counter checkoutCounter;
  CheckoutService(MeterRegistry registry) {
    this.checkoutCounter = Counter.builder("orders.checkout.count")
      .description("Number of completed checkouts")
      .register(registry);
  }
  public void completeCheckout() { checkoutCounter.increment(); }
}
```

Annotations that Instrument

Simpler Metric Creation

```
@Timed("api.response.time")
@Counted("api.requests.count")
@Observed(name = "payment.process", contextualName = "payment-service")
public Payment process(PaymentRequest request) { ... }
```

- `@Timed` measures execution time
- `@Counted` counts invocations
- `@Observed` creates metrics and optional traces via Observation API

Enable scanning of observation annotations:

```
management.observations.annotations.enabled: true
```

Common Tags and Observation Key Values

Consistent Context Everywhere

```yaml
management:
  metrics:
    tags:
      region: eu-central-1
      stack: prod
  observations:
    key-values:
      app: checkout-service
```

- Adds low cardinality context such as region, stack, app
- Enables filtering and dashboard segmentation

Export to Prometheus

Add dependency:

```
implementation 'io.micrometer:micrometer-registry-prometheus'
```

Expose endpoint and configure Prometheus:

```
management.endpoints.web.exposure.include: prometheus
```

```
scrape_configs:
  - job_name: "spring-boot"
    metrics_path: "/actuator/prometheus"
    static_configs:
      - targets: ["app:8080"]
```

Prometheus exemplars are supported when tracing is enabled.

OTLP Metrics Export

Align with OTel Backends

```yaml
management:
  otlp:
    metrics:
      export:
        url: "https://otel.example.com:4318/v1/metrics"
        headers.authorization: "Bearer ${OTLP_TOKEN}"
```

Alternative Backends

Plug and Play Integrations

Micrometer supports many systems natively:

- **Datadog:** `micrometer-registry-datadog`

```
management.datadog.metrics.export.api-key: "YOUR_KEY"
```

- **Wavefront:** `micrometer-registry-wavefront`

```
management.wavefront.api-token: "YOUR_TOKEN"
```

- **InfluxDB / OTLP / Stackdriver / JMX / New Relic** - add registry and configure endpoint

All exporters follow `management.*.metrics.export.*` configuration pattern.

Histograms, SLO Buckets, and Exemplars

Make Dashboards Meaningful

```yaml
management:
  metrics:
    distribution:
      percentiles-histogram:
        http.server.requests: true
      slo:
        http.server.requests: 100ms,300ms,1s,3s
```

With tracing on the classpath, exemplars on `http.server.requests` link to traces in Grafana.

Grafana Visualization and Alerting

From Metrics to Insights

- **Grafana + Prometheus** is a popular open source combo
- Dashboards for:
  - JVM metrics (`jvm.memory.used` , `system.cpu.usage`)
  - HTTP latencies & errors (`http.server.requests`)
  - Business KPIs (`orders.checkout.count`)
- Example alert: CPU usage > 90% for 5 min results in Slack notification

Use **common tags** for region and stack filtering.

Distributed Tracing with OpenTelemetry

Connecting the Dots Across Services

Why Distributed Tracing

The Missing Link in Observability

- Microservices and async messaging add network and thread hops
- Logs and metrics show *what* and *how often*, but not *where time is spent*
- Tracing links events across **threads, services, and networks**

Helps answer:

- "Why is checkout slower today?"
- "Which downstream call is responsible for latency?"
- "Where are retries or timeouts happening?"

OpenTelemetry in Spring Boot

Unified Tracing via Micrometer Tracing

- Micrometer Tracing bridges Observations to tracer implementations
- Auto-configured when tracing dependencies are present

Tracing context (`traceId`, `spanId`) propagates through:

- Spring MVC and WebFlux requests
- RestTemplate, WebClient, and RestClient
- Async tasks and reactive pipelines

Setting Up OpenTelemetry

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

```
management.tracing.sampling.probability: 1.0 # default is 0.1
management.zipkin.tracing.endpoint: "http://localhost:9411/api/v2/spans"
```

Tracing Exporters Matrix

Zipkin, OTLP, and Wavefront

```
# OTLP HTTP
management.otlp.tracing.endpoint: "http://otel-collector:4318/v1/traces"

# Zipkin
management.zipkin.tracing.endpoint: "http://zipkin:9411/api/v2/spans"

# Wavefront
management.wavefront.api-token: "${WAVEFRONT_TOKEN}"
```

Keep `micrometer-tracing-bridge-otel` in all cases.

How Tracing Works in Boot

- Spring Boot creates a **Tracer** bean through Micrometer Tracing
- Each HTTP request and each `@Observed` operation creates a **span**
- Spans are exported via Zipkin or OTLP depending on configuration

Supported setups:

- OpenTelemetry with Zipkin, Wavefront, or OTLP
- Brave with Zipkin or Wavefront

Correlation IDs in Logs

Link Logs to Traces

Default correlation uses `traceId` and `spanId` in MDC:

```
[803B448A0489F84084905D3093480352-3425F23BB2432450]
```

Customize correlation format:

```yaml
logging:
  pattern:
    correlation: "[${spring.application.name:},%X{traceId:-},%X{spanId:-}] "
  include-application-name: false
```

Custom Spans and Baggage

Add Context Where It Matters

```
@Component
class PaymentProcessor {
  private final ObservationRegistry registry;
  PaymentProcessor(ObservationRegistry registry) { this.registry = registry; }
  public void process() {
    Observation.createNotStarted("payment.task", registry)
      .lowCardinalityKeyValue("region", "eu")
      .observe(this::doPayment);
  }
}
```

Baggage propagation:

```
management.tracing.baggage.remote-fields: tenant-id
management.tracing.baggage.correlation-fields: tenant-id
```

Testing and Sampling Defaults

Be Explicit per Environment

- Default sampling probability is 0.1
- Increase sampling in testing or when investigating an incident
- Some test slices may not auto-configure tracing components; prefer application context tests for tracing behavior

Logging, Structured Logging, and Auditing

Structured Logs, Correlation, and Audit Insights

OTLP Logs and Correlation

- Micrometer Tracing injects `traceId` and `spanId` into MDC
- Use log appenders that support OTLP when shipping logs centrally

```
management.otlp.logging.endpoint: "https://otlp.example.com:4318/v1/logs"
```

Note: configure the OpenTelemetry Logback or Log4j2 appender in your logging config.

MDC and Correlation Fields

Trace and Span Propagation

Example output:

```
[803B448A0489F84084905D3093480352-3425F23BB2432450]
```

Recommended logging pattern:

```yaml
logging:
  pattern:
    correlation: "[${spring.application.name:},%X{traceId:-},%X{spanId:-}] "
  include-application-name: false
```

Use the auto-configured RestTemplateBuilder, WebClient.Builder, or RestClient.Builder to keep context propagation.

Manage Log Levels at Runtime

`/actuator/loggers`

```
GET  /actuator/loggers
POST /actuator/loggers/com.example.service
{ "configuredLevel": "DEBUG" }
```

Reset by posting `null` as `configuredLevel`.

```
management.endpoints.web.exposure.include: health,info,loggers
```

Structured Logging in Spring Boot

From Plain Text to JSON Insights

- Structured logging writes logs in machine-readable JSON
- Spring Boot supports these formats out of the box:
  - **Elastic Common Schema (ECS)**
  - **Graylog Extended Log Format (GELF)**
  - **Logstash JSON**

Enable with one property:

```
logging.structured.format.console: ecs # or gelf, logstash
logging.structured.format.file: logstash
logging.structured.json.customizer: com.patbaumgartner.logging.MyStructuredLoggingJsonMembersCustomizer
```

Customizing Structured JSON

Tune JSON Members and Stack Traces

```
logging.structured.json.exclude: log.level
logging.structured.json.rename.process.id: procid
logging.structured.json.add.corpname: mycorp
logging.structured.json.stacktrace.root: first
logging.structured.json.stacktrace.max-length: 1024
```

- Exclude or rename fields
- Add fixed custom fields for ingestion
- Limit stack trace length or include hashes for performance

Spring Boot Auditing

Security and Business Event Tracking

- With Spring Security, Actuator publishes audit events such as login success or failure and access denied
- Enable by providing an `AuditEventRepository` bean
  - `InMemoryAuditEventRepository` is fine for development
  - Provide a persistent implementation for production
- Publish custom business audits via `AuditApplicationEvent`
- Expose `/actuator/auditevents` only when intended

Best Practices on CI/CD and Kubernetes

Build Observability into Delivery

Design for Observability

- Treat observability as part of system design
- Ask: how will we know this works in production
- Instrument critical flows with `@Observed`, `@Timed`, or custom observations
- Document business KPIs and map them to metrics

Avoid Telemetry Overload

- Collect what you can act on, not everything measurable
- Focus on latency, throughput, and error rate
- Disable unused or noisy meters if needed

```
management.metrics.enable.example.remote: false
```

- Keep tags low cardinality
- Review dashboards regularly and prune

Consistent Naming and Tagging

- Metric names: lowercase and dot separated such as `service.orders.success.count`
- Span names: reflect operations such as `http.server.request` or `payment.process`
- Use consistent tags: `service`, `env`, `region`, `version`
- Maintain a shared observability contract across teams

Common Tags Everywhere

```yaml
management:
  metrics:
    tags:
      service: checkout-service
      version: 1.5.2
      region: eu-central-1
  observations:
    key-values:
      environment: prod
```

Applies consistent context across Tempo and Prometheus as well as Grafana.

Context Propagation in Async and Reactive

Trace Context Across Async Tasks

- Micrometer Context Propagation carries observation context across threads

```
spring.reactor.context-propagation: auto
```

- Applies to `@Async`, `CompletableFuture`, and Project Reactor
- Always build HTTP clients with the Boot-provided builders for propagation

Observability in CI and CD

Build-Time Validation and Automation

- Add checks in pipelines
  - `/actuator/health` must be `UP`
  - `/actuator/info` contains correct commit and version
- Include smoke tests that assert traces are created
- Manage dashboards and alerts as code
- Add gates that fail builds on regression signals

Observability in Kubernetes

Health, Scaling, and Consistency

Use Actuator endpoints for probes:

```
livenessProbe:
  httpGet: { path: /actuator/health/liveness, port: 8080 }
readinessProbe:
  httpGet: { path: /actuator/health/readiness, port: 8080 }
```

- Add `management.endpoint.health.probes.add-additional-paths=true` to expose `/livez` and `/readyz` on the main port
- Drive autoscaling from metrics such as CPU, latency, throughput
- Centralize dashboards per namespace or region with common tags

Choose Tracing Approach

Bridge, SDK, or Agent

Option	When to Use	Notes
Micrometer Tracing Bridge	Spring Boot apps using Micrometer	Auto-configured, minimal setup
OpenTelemetry SDK	Non-Spring services or custom needs	Full control over exporters and spans
OpenTelemetry Java Agent	Apps you cannot modify	Attach at runtime for no-code instrumentation

Tip: use the Agent for legacy apps and the Bridge for modern Spring services.

Secure Actuator Endpoints

Safe Exposure in Production

```
management.endpoints.web.exposure.include: health,info,prometheus
management.endpoint.health.show-details: when-authorized
```

- Restrict via Spring Security roles or IP allow lists
- Review `/env` and `/configprops` for sanitized secrets
- Consider a separate management port or internal network access

SBOM, HTTP Exchanges, and Ops Hygiene

Supply Chain Awareness and Safe Surfaces

SBOM Endpoint

Show Your Bill of Materials

- Enable `/actuator/sbom` when configured
- Consider CycloneDX generation in the build
- Useful for compliance and incident response

HTTP Exchanges Endpoint

Request and Response History

- Enable `httpexchanges` in dev and staging
- Keep retention small and secure the endpoint
- Use it to reproduce issues locally

Operational Hygiene Checklist

Things to Review Before Go-Live

- Discovery page disabled or restricted
- Access defaults set to `none`, then opt in per endpoint
- Cache TTL configured for heavy endpoints
- Heapdump and logfile gated behind auth and role checks

Wrap Up and Q&A

Connecting the Dots from Visibility to Insight

The Observability Trifecta

Three Pillars Working Together

- **Actuator - runtime visibility**
  - Inspect what your app is doing right now
  - `/health` , `/info` , `/metrics` , `/loggers` as operational windows
- **Micrometer - quantitative insight**
  - Turn behavior into measurable metrics
  - Export to Prometheus, Datadog, Wavefront, and more
  - Expose JVM, HTTP, DB, and business metrics
- **OpenTelemetry - cross-service traceability**
  - Connect requests across microservices
  - Correlate logs, metrics, and traces
  - Pinpoint latency and dependency issues

Common Pitfalls

And How to Avoid Them

- Telemetry overload - focus on actionable metrics
- Inconsistent naming or tags - standardize `service`, `version`, `region`
- No security review - do not expose `/env`, `/configprops`, or unrestricted `/metrics`
- Missing context propagation - always use Boot-provided HTTP client builders
- No ownership - observability is a team responsibility

Quick Wins

Steps You Can Take Tomorrow

- Enable `/actuator/health` in every app
- Add `micrometer-registry-prometheus` and visualize in Grafana
- Use `micrometer-tracing-bridge-otel` with Zipkin or Tempo
- Add business metrics and `@Observed` to key flows
- Secure and document observability endpoints

Further Resources

Learn and Connect

- **Spring Boot Actuator**: https://spring.io/guides/gs/actuator-service
- **Micrometer**: https://micrometer.io
- **OpenTelemetry**: https://opentelemetry.io/docs
- **Grafana Dashboards**: https://grafana.com/grafana/dashboards
- **Spring Observability Community**: Spring I/O talks, GitHub issues, Discord or Slack

Final Message

If You Cannot Observe It, You Cannot Trust It!

Observability is how your application speaks to you. Listen early and often.

Encourage your team to ship one real service with solid observability, visualize it, and share what you learn.

Q&A

Let's Talk About Your Context!

Thank You!

Let’s Keep Observing!

Spring Boot Observability in Practice

Actuator, Micrometer, and OpenTelemetry

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com

bitly