Observability Workshop

Spring I/O
2024-05-29
Jonatan Ivanov & Tommy Ludwig

Hello From Us

spring

Spring Team/Micrometer

Jonatan Ivanov

Tommy Ludwig

spring

Today's Schedule

- Morning Break (20 min)
  - 10:45 - 11:05
- Lunch Break (60 min)
  - 13:00 - 14:00
- Afternoon Break (20 min)
  - 15:30 - 15:50
- Ends 5pm (ish)

spring

Today's Schedule

- Machine Setup (Java + Docker)
- Tour of the Spring Boot Applications
- HTTP Interface Clients

spring

Today's Schedule

- Observability
  - Spring and Observability
  - JDBC observability
  - Grafana + Prometheus
- Manual Instrumentation
- Questions and Answers

spring

1. The fundamentals of Application Observability, why do we need it
2. How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial
3. How Micrometer provides a unified API to instrument your code for various signals and backends
4. What is Micrometer's new Observation API and how to use it
5. What signals to watch in your own application
6. Spring Boot's built-in observability features and how to customize them
7. How to avoid common issues
8. How to integrate metrics with distributed tracing and logs
9. How to visualize and analyze observability data to identify issues and optimize performance
10. How to troubleshoot issues faster and more effectively
11. The latest developments around Observability

Hello From You

Hands Up 🙋‍♀️

- You're a Java developer?
- You've used Docker?
- You've used Spring Boot?
- You've used Observability?
- You've used OpenTelemetry?

WiFi

Spring I/O Workshops bootifulBCN24

spring

Machine Setup

Minimum System Requirements

- Git
- Docker
- Java 17 (or higher)
- Java IDE

🤦 Problems? Please let us know!

How to get help?

- **README.md**
- **HELP.md** (copy-paste grand mastery)
- A “secret” **`final`** branch 😈

```
[...]
ab4ab30 Add org property
2456f04 Initial commit
```

- **`slack.micrometer.io #springio2024`**
- Let us know! 🙋

Machine Setup

```
$ git clone https://github.com/jonatan-ivanov/springio24-observability-workshop
$ cd springio24-observability-workshop

$ java --version
$ ./mvnw --version
$ ./mvnw package

$ docker compose up
$ docker compose ps
$ docker compose down #--volumes
```

spring

🤦 Problems? Please let us know!

Tour of a Spring Boot Application

About the Dog Service Sample

- It's a very silly application 😜
- Just enough code to demo what we want!
- Spring Boot 3.3
- Java 17
- JPA (Hibernate)
- Spring MVC
- Spring Security

Checkout the Code

```
$ git clone https://github.com/jonatan-ivanov/springio24-observability-workshop
$ cd springio24-observability-workshop

$ git checkout main
$ docker compose up -d

$ ./mvnw package
```

Import into your favorite IDE

spring

🙋 Problems? Please let us know!

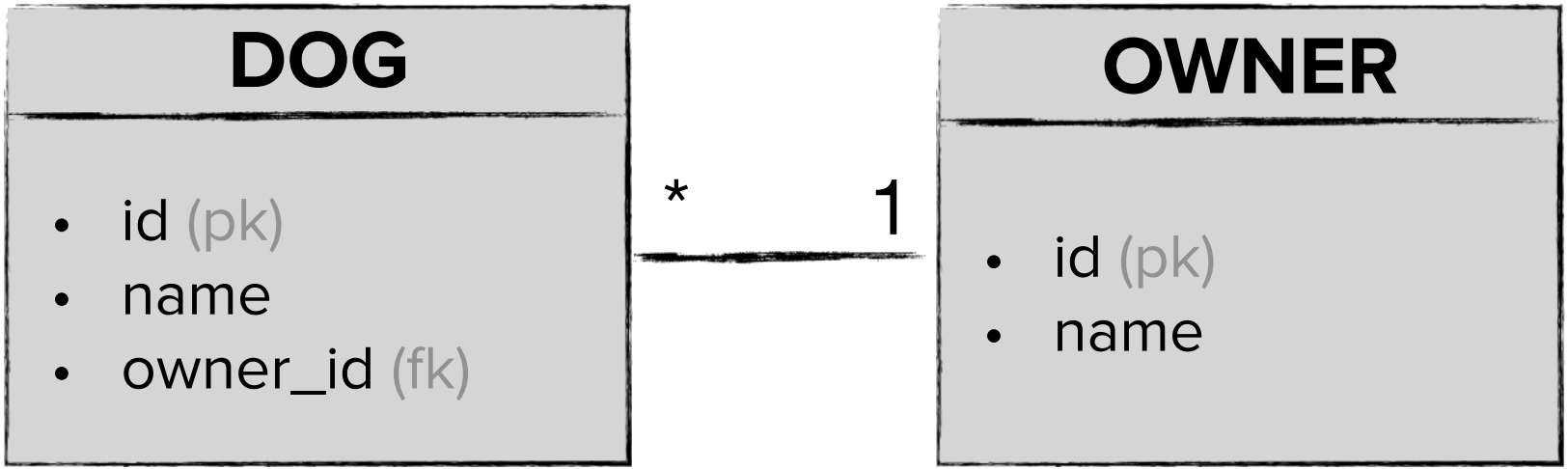
Check Out the Code - pom.xml

- Open `pom.xml`
- We're using spring-boot-starter-parent 3.3
- The `java.version` property is 17
- We're using starters for:
  - actuator, web, data-jpa, and security
- We're using PostgreSQL for the datastore

Check Out the Code - src/main/resources

Open `src/main/resources`

Look at schema.sql and data.sql files



spring

Check Out the Code - com.example.dogservice.domain

- Open `com.example.dogservice.domain` package
- `Dog` and `Owner` JPA classes map to the schema
- `DogRepository` and `OwnerRepository` are Spring Data repositories
  - `findByNameIgnoringCase` is converted to JQL automatically
- `InfoLogger` is an `ApplicationRunner` to log info at startup

Check Out the Code - com.example.dogservice.service

- Open `com.example.dogservice.service` package
- `OwnerService` uses constructor injection
- Simple facade over repositories
- Throws custom `NoSuchDogOwnerException`

Check Out the Code - com.example.dogservice.web

- Open `com.example.dogservice.web` package
- `DogsController`
  - Simple controller used for testing
- `OwnerController`
  - Delegates to the `OwnerService`
  - Deals with `NoSuchDogOwnerException`
  - Note: Meta-annotated `@RestController` and `@GetMapping`

Check Out the Code - com.example.dogservice.security

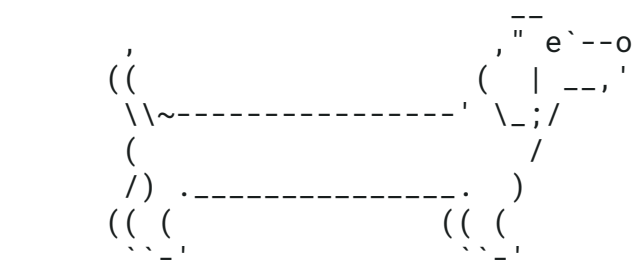
- Open `com.example.dogservice.security` package
- `SecurityConfiguration`
  - Defines our web security
- `SecurityProperties` and `UserProperties`
  - `@ConfigurationProperties` maps from values in `src/main/resources/application.yml`

Check Out the Code - src/main/resources/

- Open `src/main/resources/`
- Inspect `application.yml`
  - Defines the database connection
  - Configures JPA
  - Enables JMX
  - Configures server errors
  - Exposes all actuator endpoints
  - Enables actuators over HTTP
  - Customizes a metric name
  - Defines the in-memory user details

Run the code

```
$ ./mvnw -pl dog-service clean spring-boot:run
```



```
…
2024-04-09T10:50:22.372-05:00  INFO 151018 --- [dog-service] [           main] [
] c.example.dogservice.domain.InfoLogger   : Found owners [Tommy, Jonatan]
2024-04-09T10:50:22.386-05:00  INFO 151018 --- [dog-service] [           main] []
c.example.dogservice.domain.InfoLogger   : Found dogs [Snoopy owned by Tommy, Goofy owned by Tommy,
Clifford owned by Jonatan]
…
2024-04-09T10:50:28.260-05:00  INFO 151018 --- [dog-service] [nio-8080-exec-1] [
] o.s.web.servlet.DispatcherServlet        : Completed initialization in 2 ms
…
```

spring

Checkpoint

Everyone has a running application

Checkpoint - DogsController Works

```
$ http "http://localhost:8080/dogs"
```

Generates error JSON

```
$ http "http://localhost:8080/dogs?aregood=true"
$ http "http://localhost:8080/dogs?aregood=false"
```

```
$ http "http://localhost:8080/dogs/?aregood=false"
```

Note the trailing slash

Checkpoint - OwnerController Works

```
$ http "http://localhost:8080/owner/tommy/dogs"
```

Needs login

```
$ http -a user:password "http://localhost:8080/owner/tommy/dogs"
$ http -a user:password "http://localhost:8080/owner/jonatan/dogs"
```

```
$ http -a user:password "http://localhost:8080/owner/dave/dogs"
```

`NoSuchOwnerException` mapped to HTTP 404

Checkpoint - Actuator Works

```
$ http -a admin:secret "http://localhost:8080/actuator"
```

Open the following in a web browser:

```
http://localhost:8080/actuator
http://localhost:8080/actuator/metrics
http://localhost:8080/actuator/metrics/http.server.requests
```

Checkpoint - Actuator Over JMX Works

```
$ jconsole
```

Select `com.example.dogservice.DogServiceApplication`

Click `Connect`

Select `MBeans` on the menu

Expand org.springframework.boot, Endpoint

Checkpoint

Everyone has a working application

HTTP Client Interfaces

About the Dog Client Sample

- It's another very silly application 😛
- Just enough code to demo what we want!
- API using `HttpServiceProxy`
- Another MVC controller
- Some configuration

HTTP Client Interfaces: interfaces + metadata

- Open `com.example.dogclient.api.Api`

```
@GetExchange("/dogs")
DogsResponse dogs(@RequestParam(name = "aregood") boolean areGood);

@GetExchange("/owner/{name}/dogs")
List<String> ownedDogs(@PathVariable String name);
```

HTTP Client Interfaces + Records

- Open `com.example.dogclient.api.DogsResponse`

```
public record DogsResponse(String message) {
}
```

HTTP Client Interfaces - Building the client

- Open `com.example.dogclient.DogClientApplication`

```
WebClient webClient =
  webClientBuilder.baseUrl("…").build();
WebClientAdapter adapter =
  WebClientAdapter.create(webClient);
HttpServiceProxyFactory factory =
  HttpServiceProxyFactory.builderFor(adapter).build();

return factory.createClient(Api.class);
```

HTTP Client Interfaces - Using the client

```
// Signature
DogsResponse dogs(boolean areGood);
List<String> ownedDogs(String name);

// Usage
api.dogs(true);
api.ownedDogs("Tommy");
```

ApplicationRunner

- Open `com.example.dogclient.DogClientApplication`
- Prints information when the application starts
- Functional interface called when the app starts:

```
void run(ApplicationArguments args) throws Exception
```

Run the application - Check the startup output

```
$ ./mvnw -pl dog-service clean spring-boot:run
```

```
$ ./mvnw -pl dog-client clean spring-boot:run
```

```
2024-04-09T13:26:07.471-05:00  INFO 1618241 --- [dog-client] [           main] [
] c.e.dogclient.DogClientApplication       : Started DogClientApplication in 1.505
seconds (process running for 1.643)

DogsResponse[message=We <3 dogs!!!]

[Snoopy, Goofy]
```

```
$ http "http://localhost:8081/owner/tommy/dogs"
```

Checkpoint

Client application works

Observability

What is Observability?

What is Observability?

How well we can understand the internals of a system based on its outputs

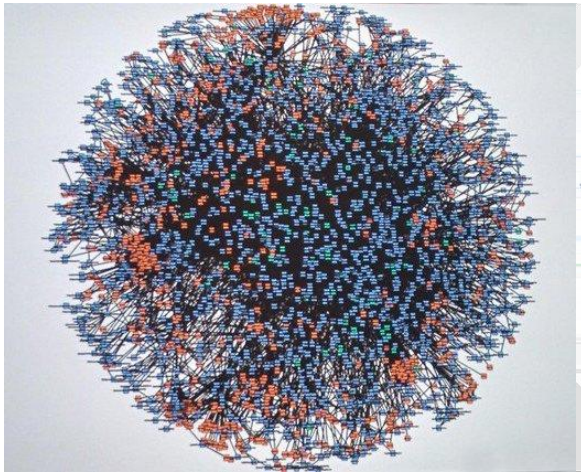
(Providing ***meaningful*** information about what happens inside)
(Data about your app)

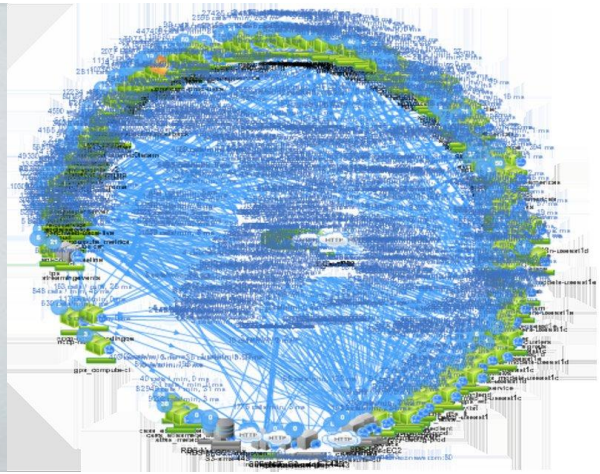
Why do we need Observability?

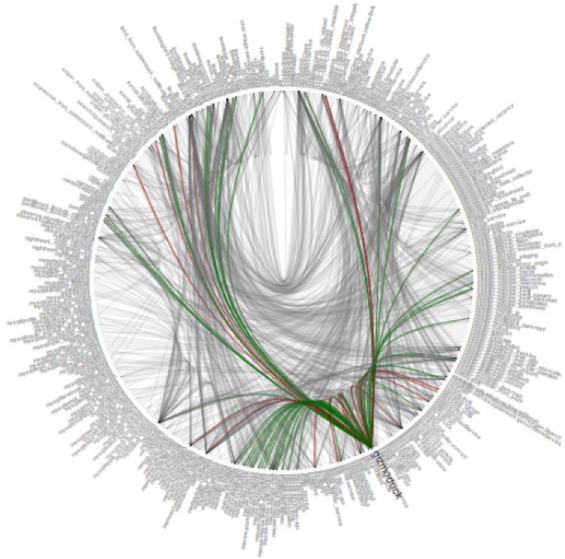
Why do we need Observability?

Today's systems are increasingly complex (cloud)

(Death Star Architecture, Big Ball of Mud)







spring

Why do we need Observability?

Environments can be chaotic

You turn a knob here a little and apps are going down there

We need to deal with unknown unknowns

We can’t know everything

Things can be perceived differently by observers

Everything is broken for the users but seems ok to you

Why do we need Observability? (business perspective)

Reduce lost revenue from production incidents

Lower mean time to recovery (MTTR)

Require less specialized knowledge

Shared method of investigating across system

Quantify user experience

Don't guess, measure!

Logging
Metrics
Distributed Tracing

Logging - Metrics - Distributed Tracing

Logging

What happened (why)?

Emitting events

Metrics

What is the context?

Aggregating data

Distributed Tracing

Why happened?

Recording causal ordering of events

Examples

Latency

Logging

HTTP request took 140ms

Metrics

P99.999: 140ms

Max: 150 ms

Distributed Tracing

DB was slow

(lot of data was requested)

Error

Logging

Request failed (stacktrace?)

Metrics

The error rate is 0.001/sec

2 errors in the last 30 minutes

Distributed Tracing

DB call failed

(invalid input)

Checkpoint

Everyone knows what Observability is

1. **The fundamentals of Application Observability, why do we need it**
2. How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial
3. How Micrometer provides a unified API to instrument your code for various signals and backends
4. What is Micrometer's new Observation API and how to use it
5. What signals to watch in your own application
6. Spring Boot's built-in observability features and how to customize them
7. How to avoid common issues
8. How to integrate metrics with distributed tracing and logs
9. How to visualize and analyze observability data to identify issues and optimize performance
10. How to troubleshoot issues faster and more effectively
11. The latest developments around Observability

Logging
with JVM/Spring

Logging with JVM/Spring: SLF4J + Logback

SLF4J with Logback comes pre-configured

SLF4J (Simple Logging Façade for Java)

Simple API for logging libraries

Logback

Natively implements the SLF4J API

If you want Log4j2 instead of Logback:

```
- spring-boot-starter-logging
+ spring-boot-starter-log4j2
```

Setup Logging - Add org property

- We will need something that we can use to query:
  - All of our apps (**`spring.application.org`**)
  - Only one app (**`spring.application.name`**)
  - Only one instance (we only have one instance/app)

```
spring:
  application:
    name: dog-service
    org: petclinic
```

Setup Centralized Logging - Add Loki4J

- Copy
  From: **`dog-client/src/main/resources/logback-spring.xml`**
  To: **`dog-service/src/main/resources/logback-spring.xml`**
- Add dependency to **`pom.xml`**

```
<dependency>
    <groupId>com.github.loki4j</groupId>
    <artifactId>loki-logback-appender</artifactId>
    <version>1.5.1</version>
</dependency>
```

spring

Setup Logging - Do we have logs?

- Go to Grafana: `http://localhost:3000`
- Choose Explore, then Loki from the drop down
- Search for `application = dog-service`
- Search for `org = petclinic`
- We will get back to our logs later

Checkpoint

Everyone has logs in Loki for both services

Coffee break (20 minutes)

10:45 - 11:05

Metrics
with JVM/Spring

Metrics with JVM/Spring: Micrometer

Dimensional Metrics library on the JVM

Like SLF4J, but for metrics

API is independent of the configured metrics backend

Supports many backends

Comes with `spring-boot-actuator`

Spring projects are instrumented using Micrometer

Many third-party libraries use Micrometer

Supported metrics backends/formats/protocols

AppOptics

Atlas

Azure Monitor

CloudWatch (AWS)

Datadog

Dynatrace

Elastic

Ganglia

Graphite

Humio

InfluxDB

JMX

KairosDB

New Relic

(/actuator/metrics)

OpenTSDB

OTLP

Prometheus

SignalFx

Stackdriver (GCP)

StatsD

Wavefront (VMware)

Setup Metrics

Setup Metrics - Add the org to Observations and Metrics

application.yml

```
management:
  observations:
    key-values:
      org: ${spring.application.org}
  metrics:
    tags:
      application: ${spring.application.name}
      org: ${spring.application.org}
```

Setup Metrics - Let's check Metrics 🧐

- `http://localhost:8080/actuator/prometheus`
- `401` 🧐
- Prometheus? `http://localhost:9090/targets`
- Spring Security! 👀
- Let's disable it, what could go wrong!? 😈
- Everyone, please don't do this in prod!
- Unless you want everyone to know about it. 😈

Setup Metrics - Disable auth for certain endpoints

SecurityConfiguration.java

```
requests
  .requestMatchers("/dogs", "/actuator/**").permitAll();
```

Setup Metrics - Add histogram support for http metrics

- We want to see the latency distributions on our dashboards
- We want to calculate percentiles (tp99?)

```
management:
  metrics:
    distribution:
      percentiles-histogram:
        # all: true
        http.server.requests: true
```

spring

Setup Metrics - Let’s check the HTTP and JVM metrics

- Let’s check **`/actuator/metrics`**
  `/actuator/metrics/{metricName}`
  `/actuator/metrics/{metricName}?tag=key:value`
- Let’s write a Prometheus query (**`HELP.md`**)

```
sum by (application) (rate(http_server_requests_seconds_count[5m]))
```

- Let’s check the dashboards: go to Grafana, then Browse
  - Spring Boot Statistics
  - Dogs

Checkpoint

Everyone has metrics on the dashboards

Distributed Tracing with JVM/Spring

Distributed Tracing with JVM/Spring 🚀

Boot 2.x: `Spring Cloud Sleuth`

Boot 3.x: `Micrometer Tracing`
(Sleuth w/o Spring dependencies)

Provide an abstraction layer on top of tracing libraries

- Brave (OpenZipkin), default
- OpenTelemetry (CNCF), experimental

Instrumentation for Spring Projects, 3rd party libraries, your app

Support for various backends

Setup Distributed Tracing - Add Micrometer Tracing

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.reporter2</groupId>
  <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
```

Setup Distributed Tracing - Set sampling probability 🧐

```
management:
  tracing:
    sampling:
      probability: 1.0
```

spring

Setup Distributed Tracing - Setup log correlation 🚀

- If you are on Spring Boot 3.1 or above, this is not needed
- If you are on 3.1 or lower, you need to set `logging.pattern.level`
- We are on 3.3!

```
logging:
  level:
    org.springframework.web.servlet.DispatcherServlet: DEBUG
```

Setup Distributed Tracing - Let’s look at correlated logs

```
2024-03-22T20:13:21.588Z
DEBUG
2167
---
[dog-service]
[http-nio-8090-exec-5]
[65fde66134624d949e80e0d3241ed138-9e80e0d3241ed138]
o.s.web.servlet.DispatcherServlet: Completed 200 OK
```

Setup Distributed Tracing - Let’s look at some traces

- Go to Grafana, then Explore and choose Tempo
- Terminology
  - Span
  - Trace
  - Tags
  - Annotations

Checkpoint

Everyone has log correlation and traces in Tempo

1. The fundamentals of Application Observability, why do we need it
2. **How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial**
3. How Micrometer provides a unified API to instrument your code for various signals and backends
4. What is Micrometer's new Observation API and how to use it
5. What signals to watch in your own application
6. **Spring Boot's built-in observability features** and how to customize them (to be continued…)
7. How to avoid common issues
8. How to integrate metrics with distributed tracing and logs
9. How to visualize and analyze observability data to identify issues and optimize performance
10. How to troubleshoot issues faster and more effectively
11. The latest developments around Observability (to be continued…)

Observation API

You want to instrument your application...

- Add logs
  (application logs)
- Add metrics
  - Increment Counters
  - Start/Stop Timers
- Add Distributed Tracing
  - Start/Stop Spans
  - Log Correlation
  - Context Propagation

Instrumentation with the Observation API 🚀

```
Observation observation = Observation.start("talk",registry);
try { // TODO: scope
    doSomething();
}
catch (Exception exception) {
    observation.error(exception);
    throw exception;
}
finally { // TODO: attach tags (key-value)
    observation.stop();
}
```

spring

Configuration with the Observation API 🚀

```
ObservationRegistry registry = ObservationRegistry.create();

registry.observationConfig()
    .observationHandler(new MeterHandler(...))
    .observationHandler(new TracingHandler(...))
    .observationHandler(new LoggingHandler(...))
    .observationHandler(new AuditEventHandler(...));
```

Observation API 🚀

```
Observation.createNotStarted("talk",registry)
    .lowCardinalityKeyValue("event", "SIO")
    .highCardinalityKeyValue("uid", userId)
    .observe(this::talk);

@Observed
```

spring

1. The fundamentals of Application Observability, why do we need it
2. How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial
3. How Micrometer provides a unified API to instrument your code for various signals and backends (to be continued…)
4. **What is Micrometer's new Observation API and how to use it**
5. What signals to watch in your own application
6. **Spring Boot's built-in observability features** and how to customize them (to be continued…)
7. How to avoid common issues
8. How to integrate metrics with distributed tracing and logs
9. How to visualize and analyze observability data to identify issues and optimize performance
10. How to troubleshoot issues faster and more effectively
11. The latest developments around Observability (to be continued…)

Interoperability

Setup Observations - Disable Spring Security Observations

SecurityConfiguration.java

```
@Bean
ObservationPredicate noSpringSecurityObservations() {
 return (name,ctx)-> !name.startsWith("spring.security.");
}
```

Setup Observations - Disable Actuator Observations

ActuatorConfiguration.java

```
if (name.equals("http.server.requests") &&
  ctx instanceof ServerRequestObservationContext sc) {
    return !sc.getCarrier()
      .getRequestURI()
      .startsWith("/actuator");
} else return true;
```

Setup Observations - Enable JDBC Observations 🚀

Tadaya Tsuyukubo 😎

`net.ttddyy.observation:datasource-micrometer-spring-boot`

(1.0.3)

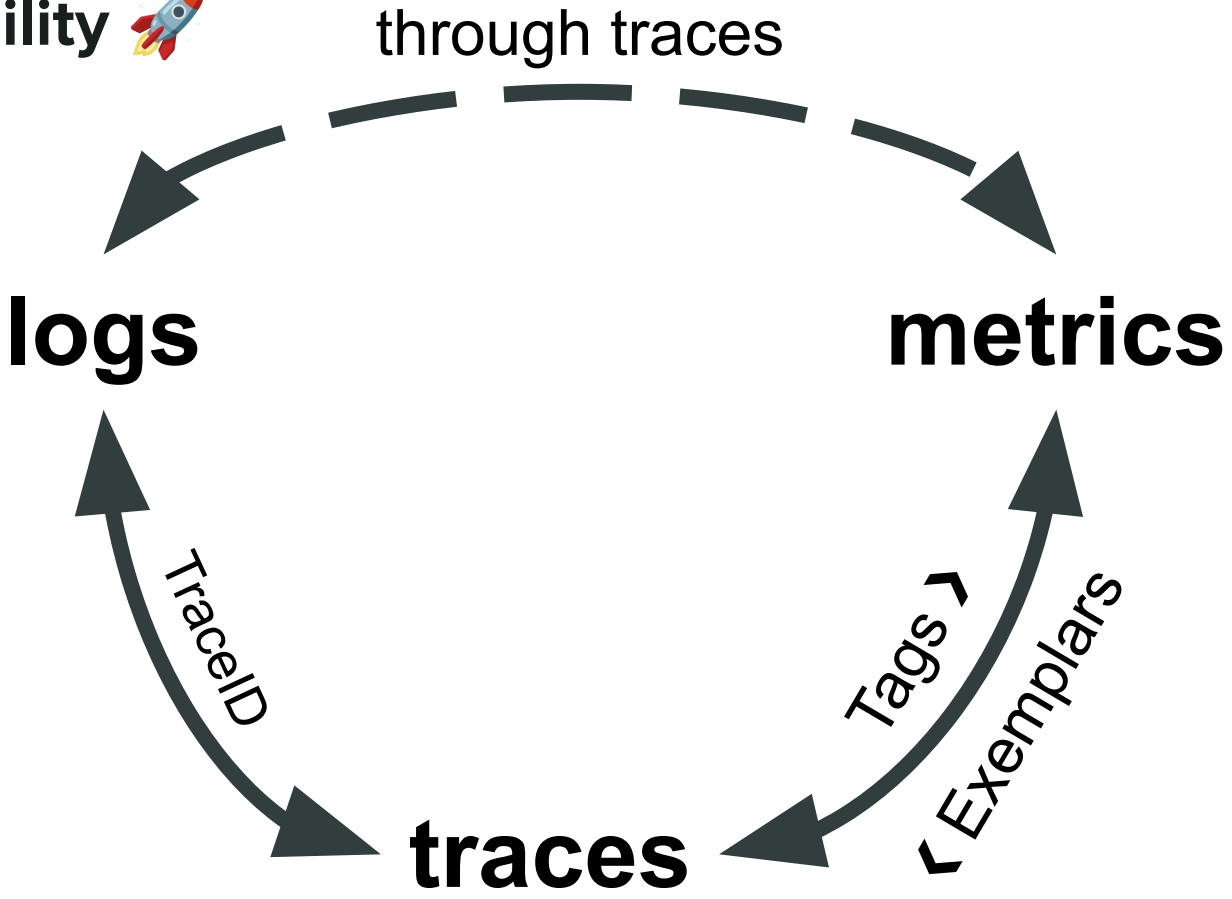
```
jdbc:
  datasource-proxy:
    include-parameter-values: true
    query:
      enable-logging: true
      log-level: INFO
```

Setup Observations - Add custom Observation

OwnerService.java

```
Observation.createNotStarted("getDogs", registry)
  .contextualName("gettingOwnedDogs")
  .highCardinalityKeyValue("owner", owner)
  .observe(() -> {
    //…
  });
```

Interoperability 🚀



spring

Interoperability - How to check Exemplars

- Exemplars are only available if you request the OpenMetrics format
- Your browser does not do this

```
http :8081/actuator/prometheus /
'Accept: application/openmetrics-text;version=1.0.0'
| grep trace_id
```

Checkpoint

Logs <=> Metrics <=> Traces

Setup Observations - Log error and signal it

OwnerController.java

```
ProblemDetail onNoSuchDogOwner(
  HttpServletRequest request,
  NoSuchDogOwnerException ex) {
  logger.error("Ooops!", ex);
  ServerHttpObservationFilter
    .findObservationContext(request)
    .ifPresent(context -> context.setError(ex));
```

Setup Observations - Hack error reporting for Tempo

```
ObservationFilter tempoErrorFilter() {
  return context -> {
    if (context.getError() != null) {
      context.addHighCardinalityKeyValue(
        KeyValue.of("error", "true")
      );
      context.addHighCardinalityKeyValue(
        KeyValue.of(
          "errorMsg", context.getError().getMessage())
      );
    }
    return context;
  };
}
```

Setup Observations - Hack DB tags for ServiceGraph

```
if(ctx instanceof DataSourceBaseContext dsCtx){
  ctx.addHighCardinalityKeyValue(
    KeyValue.of(
      "db.name", dsCtx.getRemoteServiceName()
    )
  );
}
```

spring

Actuator - Add Java, OS, and Process InfoContributors

```
management:
  info:
    java:
      enabled: true
    os:
      enabled: true
    process:
      enabled: true
```

Checkpoint

The applications are observable! 😎

1. The fundamentals of Application Observability, why do we need it
2. How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial
3. How Micrometer provides a unified API to instrument your code for various signals and backends (to be continued…)
4. What is Micrometer's new Observation API and how to use it
5. What signals to watch in your own application (to be continued…)
6. **Spring Boot's built-in observability features and how to customize them**
7. How to avoid common issues
8. **How to integrate metrics with distributed tracing and logs**
9. **How to visualize and analyze observability data to identify issues and optimize performance**
10. **How to troubleshoot issues faster and more effectively**
11. The latest developments around Observability (to be continued…)

Lunch break (1 hour)

13:00 - 14:00

Manual Instrumentation

Micrometer: MeterRegistry

- **`Meter`**: interface to collect measurements
- **`MeterRegistry`**: abstract class to create/store **`Meter`**s
- Backends have implementations of **`MeterRegistry`**
- **`SimpleMeterRegistry`** (debugging, testing, actuator)
- **`SimpleMeterRegistry#getMetersAsString`**
- **`CompositeMeterRegistry`**

Dimensionality

- Dimensional vs. Hierarchical
- **Dimensional**: metrics enriched with key/value pairs
- **Hierarchical**: key/value pairs flattened, added to the name

Dimensionality - Dimensional Example (Prometheus)

```
http_server_requests_count{
    application="tea-service",
    exception="None",
    method="GET",
    outcome="SUCCESS",
    profiles="local",
    status="200",
    uri="/tea/{name}"} 2.0
```

spring

Dimensionality - Hierarchical Example

```
"http-server-requests-count.tea-service.None
.GET.SUCCESS.local.200./tea/{name}": 2.0
```

Cumulative vs. Delta (temporality)

Cumulative: the reported value is the total value since the beginning of the measurements

Delta: the reported value is the difference in the measurements since the last time it was reported

Cumulative vs. Delta (temporality) - Example

- We count certain events and report these every minute
- The event happened 3 times in the first minute, 2 times in the second, and once in the third
- Cumulative says: 3, 5, 6 (running total)
- Delta says: 3, 2, 1 (difference)

Push vs. Poll

Poll: the backend polls the apps for metrics at their leisure (e.g.: Prometheus)

Push: the apps send metrics to the backend on a regular interval (e.g.: InfluxDB, ElasticSearch, etc.)

Creating a MeterRegistry

```
PrometheusMeterRegistry registry =
new PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
// [...]
System.out.println(registry.scrape());
```

Micrometer basic Meter types - example use case

- Counter - cache hits
- Gauge - CPU usage %
- Timer - HTTP server request timing
- DistributionSummary - HTTP request size
- LongTaskTimer - timing batch job processing

Micrometer: Counter

- Records a single metric: a count
- Monotonic: only **`increment()`**, no **`decrement()`**
- Example: number of cache hits

```
Counter counter = registry.counter("test");
counter.increment();
```

Micrometer: Counter + Tags

```
registry.counter(
   "test.counter",
   "application", "test"
).increment();
```

```
test_counter_total{application="test"} 1.0
```

Micrometer: Counter + Builder

```
Counter.builder("test.counter")
        .description("Test counter")
        .baseUnit("events")
        .tag("application", "test")
        .register(registry) // create or get
        .increment();
```

High Cardinality 🧐

```
for (int i = 0; i < 100000; i++) {
   Counter.builder("test.counter")
         .tag("userId", String.valueOf(i))
         .register(registry)
         .increment();
}
```

High Cardinality 🧐

- **`userId`** (lots of users)
- **`email`** (lots of users)
- any **`resourceId`** (lots of resources)
- **`requestId/txId/traceId/spanId/etc.`**
- Request URL
- Any (unsanitized) user input
- Please always sanitize/normalize any user input!
- Otherwise: DoS 😞

High Cardinality 🧐

High cardinality should be avoided whenever possible. In cases where it isn't possible to avoid it, see:

- `MeterFilter.maximumAllowableMetrics(...);`
- `MeterFilter.maximumAllowableTags(...);`
- `MeterFilter.ignoreTags(...);`
- `HighCardinalityTagsDetector`

Micrometer: Gauge

- A handle to get the current value
- Non-monotonic: can increase and decrease
- **“Asynchronous”**
- “Heisen-Gauge”
- “State” should be mutable and “referenced”
- Examples: queue size, number of threads, CPU temperature

Never gauge something you can count with a **`Counter`**!

Micrometer: Gauge

```
private final AtomicLong value =
   new AtomicLong(); // mutable + referenced


// elsewhere “register”
registry.gauge("test", value);



value.set(2); // elsewhere update the value
```

Micrometer: Gauge 🧐

```
// immutable :(
private Double value = 1024.0;

registry.gauge("test", value);
value = 1.0;
System.out.println(registry.scrape());
// ???
```

spring

Micrometer: Gauge 🧐

```
// not referenced either :(
private Double value = 1024.0;

registry.gauge("test", value);
value = 1.0;
System.gc(); // well…
System.out.println(registry.scrape());
```

Micrometer: Gauge

```
private final AtomicLong value =
   registry.gauge("test", new AtomicLong());
```

```
value.set(4); // elsewhere update the value
```

Micrometer: Gauge

```
private final List<String> list =
   new ArrayList<>();


registry.gauge("test", list, List::size);



list.add("test");
```

Micrometer: Gauge

```
private final List<String> list =
   registry.gauge(
      "test",
      Tags.empty(),
      new ArrayList<>(), // “state object”
      List::size         // “value function”
   );
```

spring

Micrometer: Gauge

```
private final List<String> list =
   registry.gaugeCollectionSize(
      "test",
      Tags.empty(),
      new ArrayList<>()
   );
```

Micrometer: Gauge

```
private final Map<String, Integer> map =
   registry.gaugeMapSize(
      "test",
      Tags.empty(),
      new HashMap<>()
   );
```

Micrometer: Gauge

```
private final TemperatureSensor sensor =
   new TemperatureSensor();

Gauge.builder(
   "test",
   () -> sensor.getTemperature() - 273.15
).register(registry);
```

Micrometer: Gauge

```
private final TemperatureSensor sensor =
    new TemperatureSensor();

Gauge.builder(
    "test",
    sensor::getTemperature
).register(registry);
```

Micrometer: DistributionSummary

- Tracks the distribution of recorded values
- It has one method: **`record(amount)`**
- Always reports **`count, sum, max`**
- Can report: Histograms, SLOs, and Percentiles
- Example: payload sizes of requests and responses

Micrometer: DistributionSummary

```
DistributionSummary ds = DistributionSummary
   .builder("response.size")
   .baseUnit("bytes")
   .register(registry);


ds.record(10);
ds.record(20);
```

Micrometer: Timer

- Tracks the latency of events
- Like **`DistributionSummary`** but the unit is time
- Multiple ways to record latency
- Always reports **`count, sum, max`**
- Can report: Histograms, SLOs, and Percentiles
- Example: processing time of incoming requests

Never count something that you can time with a **`Timer`** or summarize with a **`DistributionSummary`**!

count, sum, max

`count`: same as having a **`Counter`** (rate)

`sum`: sum of the recorded values (**`sum/count`**?)

`max`: max of the recorded values (time-windowed)

See the note in this section of the docs.

Micrometer: Timer

```
Timer timer = Timer.builder("requests")
   .register(registry);
```

```
Sample sample = Timer.start();
doSomething();
sample.stop(timer);
```

Micrometer: Timer

```
timer.record(() -> doSomething());
timer.recordCallable(() -> getSomething());

Runnable r = timer.wrap(() -> doSomething());
Callable c = timer.wrap(() -> getSomething());
```

Micrometer: Timer 🧐

```
// Don't do this! If you do, use nanoTime() 🙏


long start = System.nanoTime();
doSomething();
long end = System.nanoTime();
timer.record(end - start, NANOSECONDS);
```

spring

Micrometer: Clock 🧐

- There is a **`Clock`** abstraction in Micrometer
- **`wallTime`** [ms]: for the current time, not for elapsed time
- **`monotonicTime`** [ns]: for measuring elapsed time
- Testing: **`MockClock`**, you can set the time with it 😈
  no **`Thread.sleep(...)`**

Coffee break (20 minutes)

15:30 - 15:50

Micrometer: LongTaskTimer

- “ActiveTaskTimer” 🧐
- Tracks the elapsed time of active events
- **`Timer`** records latency after the events finished
- **`LongTaskTimer`** records latency of running events
- **`Timer`**: past, **`LongTaskTimer`**: present
- Always reports **`count, sum, max`**
- Can report: Histograms, SLOs, and Percentiles
- Example: processing time of in-progress requests

Micrometer: LongTaskTimer

```
LongTaskTimer ltt = LongTaskTimer
    .builder("test")
    .register(registry);


Sample sample = ltt.start();
doSomething();
sample.stop();
```

Micrometer: LongTaskTimer

```
ltt.record(() -> doSomething());
ltt.recordCallable(() -> getSomething());
```

Micrometer: (Client-Side) Percentiles 🧐

- **`Timer, DistributionSummary, LongTaskTimer`**
- **Approximated** on the client side
- Not aggregatable and only percentiles configured up-front are available
- Use Histogram instead if you can

```
Timer.builder("requests")
        .publishPercentiles(0.99, 0.999)
        .register(registry);
```

spring

Micrometer: (Percentile) Histogram

- **`Timer, DistributionSummary, LongTaskTimer`**
- Show the “frequency” of values in a certain range
- Arbitrary percentiles are approximated on the backend
- Aggregatable!

```
Timer.builder("requests")
        .publishPercentileHistogram()
        .register(registry);
```

Micrometer: SLOs

- **`Timer, DistributionSummary, LongTaskTimer`**
- Additional histogram “buckets”
- Specific thresholds so you can count recordings above/below the threshold

```
Timer.builder("requests")
.serviceLevelObjectives(Duration.ofMillis(10))
.register(registry);
```

spring

Micrometer: MeterProvider 🚀

```
MeterProvider<Counter> provider =
Counter.builder("test")
      .tag("static", "42")
      .withRegistry(registry);


provider.withTags("dynamic",value).increment();
```

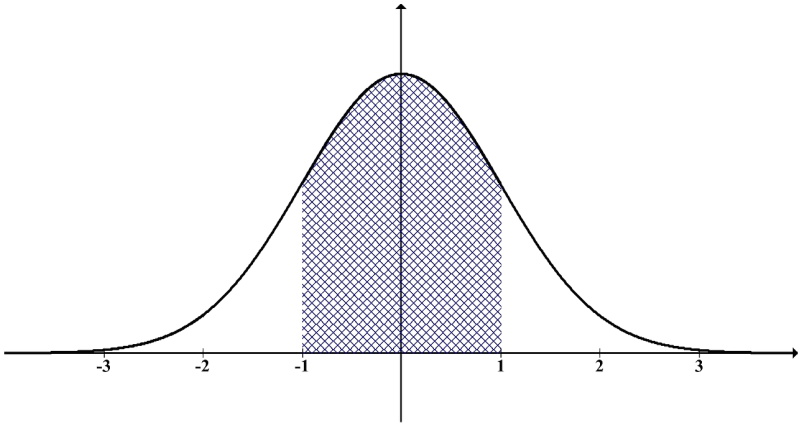
avg, tp95/99, max 🧐

- avg(7,9,12,8,10,11,8,5,168,182) = 42
- $(1 - 0.95^{42}) * 100\% = 88.40\%$
- avg(tp95) = 😬
- tp95(tp95) = 😬
- max = 😉

Latency Expectations vs. Reality 🧐

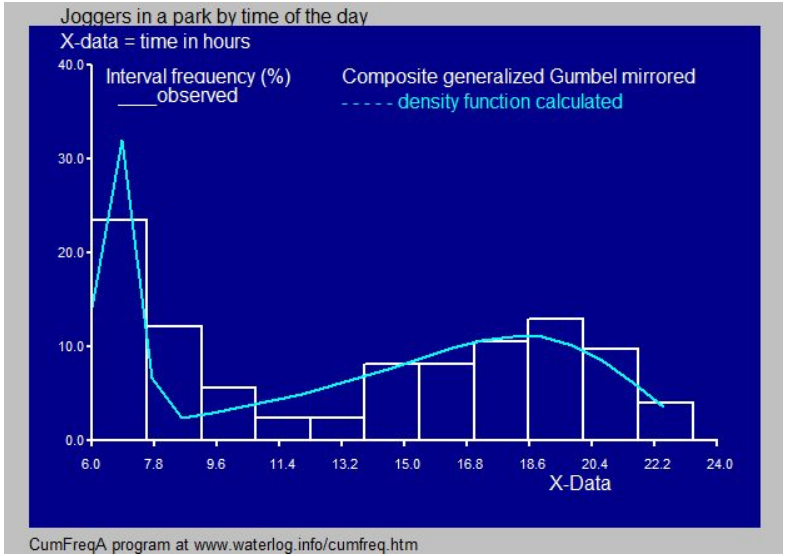
Expectation

- Normal distribution



Reality

- Long tail distribution
- Multi-modal distribution



https://commons.wikimedia.org/wiki/File:Joggers.png

Alerting

- Don’t stare at dashboards; use alerts
- Base alerts on metrics that represent business impact
- Avoid duplicate alerts
- Don’t alert on average latency!
- See previous slide about SLOs

Micrometer: MeterFilter

- Deny or Accept Meters
- Transform Meter IDs (name, tags, description, unit)
- Configure Distribution Statistics
- Separates instrumentation from configuration

Micrometer: MeterFilter

```
registry.config()
.meterFilter(MeterFilter.commonTags(...))
.meterFilter(MeterFilter.ignoreTags(...))
.meterFilter(MeterFilter.renameTag(...))
.meterFilter(MeterFilter.replaceTagValues(...))
.meterFilter(MeterFilter.denyNameStartsWith(...))
.meterFilter(MeterFilter.acceptNameStartsWith(...));
```

Micrometer: MeterFilter

```
.meterFilter(new MeterFilter() {
   @Override
   public Id map(Id id) {
      if (id.getName().equals("old"))
         return id.withName("new");
      else
         return id;
   }
});
```

Micrometer Tracing: Span

```
Span span = tracer.nextSpan().name("test");
try (SpanInScope ws = tracer.withSpan(span.start())) {
   span.tag("userId", userId);
   span.event("logout");
}
finally {
   span.end();
}
```

Micrometer: Observation 🚀

```
ObservationRegistry registry =
    ObservationRegistry.create();
registry.observationConfig().observationHandler(...);


Observation observation =
    Observation.createNotStarted("talk",registry)
        .contextualName("talk observation")
        .lowCardinalityKeyValue("event", "SIO")
        .highCardinalityKeyValue("uid", userId);
```

spring

Micrometer: Observation 🚀

```
try (Scope scope = observation.start().openScope()) {
    doSomething();
    observation.event(Event.of("question"));
}
catch (Exception exception) {
    observation.error(exception);
    throw exception;
}
finally {
    observation.stop();
}
```

spring

Micrometer: ObservationPredicate 🚀

Should the Observation be created or ignored (noop)?

```
registry.observationConfig()
.observationPredicate(
   (name, ctx) -> !name.startsWith("ignored")
);
```

Micrometer: ObservationFilter 🚀

Modify the Context

```
registry.observationConfig()
.observationFilter(
   ctx -> ctx.addLowCardinalityKeyValue(...)
);
```

Checkpoint

Manual instrumentation works! 😎

What's ~new?

- Micrometer Tracing (Sleuth w/o Spring deps.)
- Micrometer Docs Generator
- Micrometer Context Propagation
- Observation API
- Exemplars
- OTLP
- Prometheus 1.x

1. The fundamentals of Application Observability, why do we need it
2. How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial
3. **How Micrometer provides a unified API to instrument your code for various signals and backends**
4. What is Micrometer's new Observation API and how to use it
5. **What signals to watch in your own application**
6. Spring Boot's built-in observability features and how to customize them
7. **How to avoid common issues**
8. How to integrate metrics with distributed tracing and logs
9. How to visualize and analyze observability data to identify issues and optimize performance
10. How to troubleshoot issues faster and more effectively
11. **The latest developments around Observability**

Q&A

1. The fundamentals of Application Observability, why do we need it
2. How to apply these fundamentals to realistic scenarios in sample applications where having observability is crucial
3. How Micrometer provides a unified API to instrument your code for various signals and backends
4. What is Micrometer's new Observation API and how to use it
5. What signals to watch in your own application
6. Spring Boot's built-in observability features and how to customize them
7. How to avoid common issues
8. How to integrate metrics with distributed tracing and logs
9. How to visualize and analyze observability data to identify issues and optimize performance
10. How to troubleshoot issues faster and more effectively
11. The latest developments around Observability

Thank you!

slack.micrometer.io #springio2024