SpringOne Platform by Pivotal

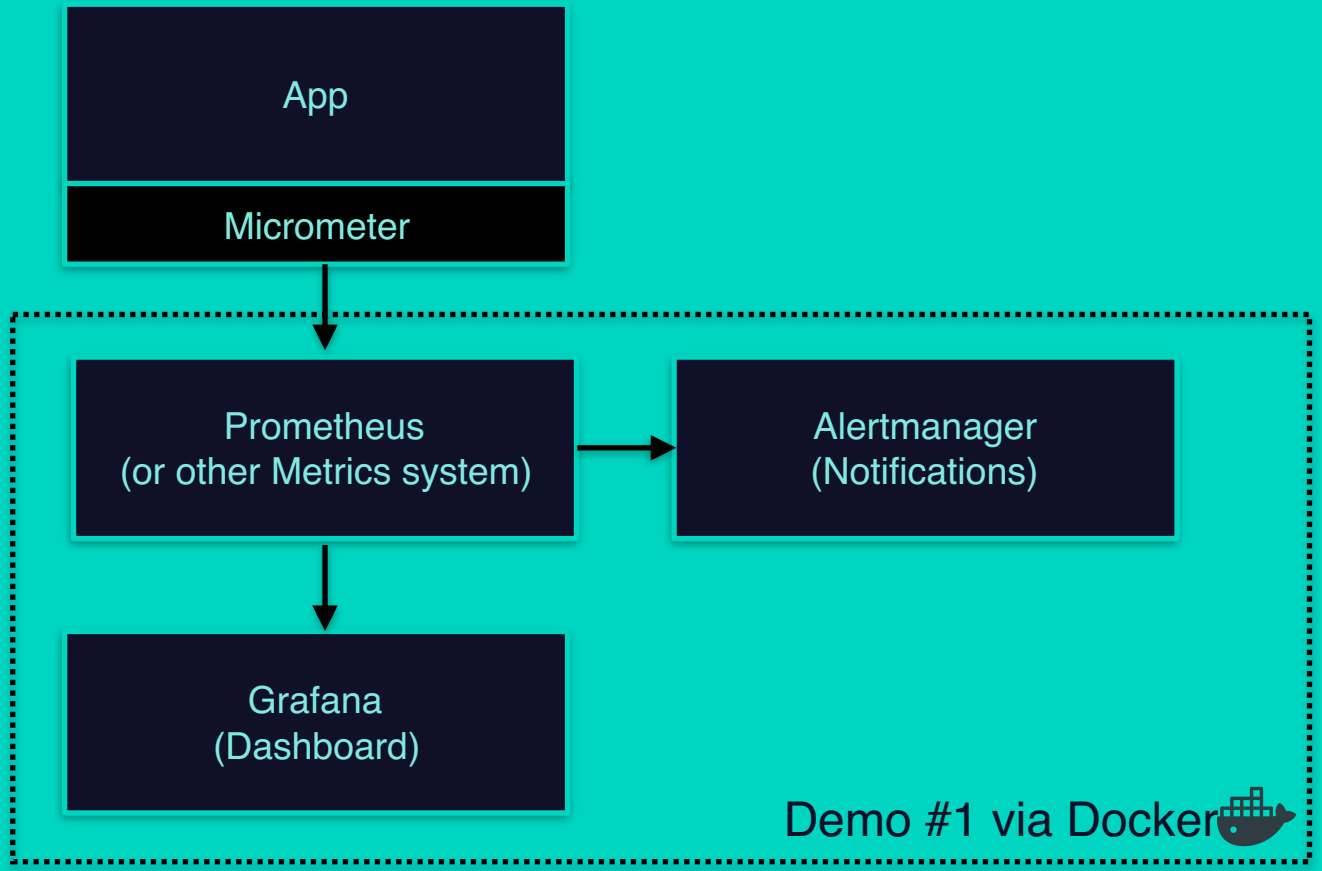
Performance Monitoring Backend and Frontend Using Micrometer

Clint Checketts - @checketts
Church of Jesus Christ of Latter-day Saints

October 7–10, 2019
Austin Convention Center

Presentation Topics

- How can I use Micrometer?
- How can I control the number of metrics I create (due to costs for my metric platform)
- I'm currently using X for metrics, how can I use Micrometer to keep using X while transitioning to hot new Y?
- How can I ensure my metrics include common information of that cluster/region/team/etc?
- And more!



Dimensional Metrics

Micrometer provides vendor-neutral interfaces for **timers**, **gauges**, **counters**, **distribution summaries**, and **long task timers** with a dimensional data model that, when paired with a dimensional monitoring system, allows for efficient access to a particular named metric with the ability to drill down across its dimensions.

Dimensional Metrics versus Hierarchical

Hierarchical:

```
server1.http.requests = 10
```

Dimensional:

```
http_requests{server="server1"} 10
```

What if I want to track by cluster or region?

How about uri or response code?

Or if I want to add metadata to a metric upon collection?

Dimensional Metrics versus Hierarchical

Hierarchical:

```
server1.http.requests = 10
us-east.blue.server1.http.requests.200.users = 10
```

Dimensional:

```
http_requests{server="server1"} 10
http_requests{server="server1", region="us-east",
  cluster="blue", status="200", uri="users"} 10
```

Monitoring for errors

versus

understanding the system

Observability:

1. Logging
2. Metrics
3. Tracing

Observability Definitions:

1. Logging
    - Detailed information about individual actions
2. Metrics
    - Aggregate information about application features
3. Tracing
    - Sampled information across multiple services

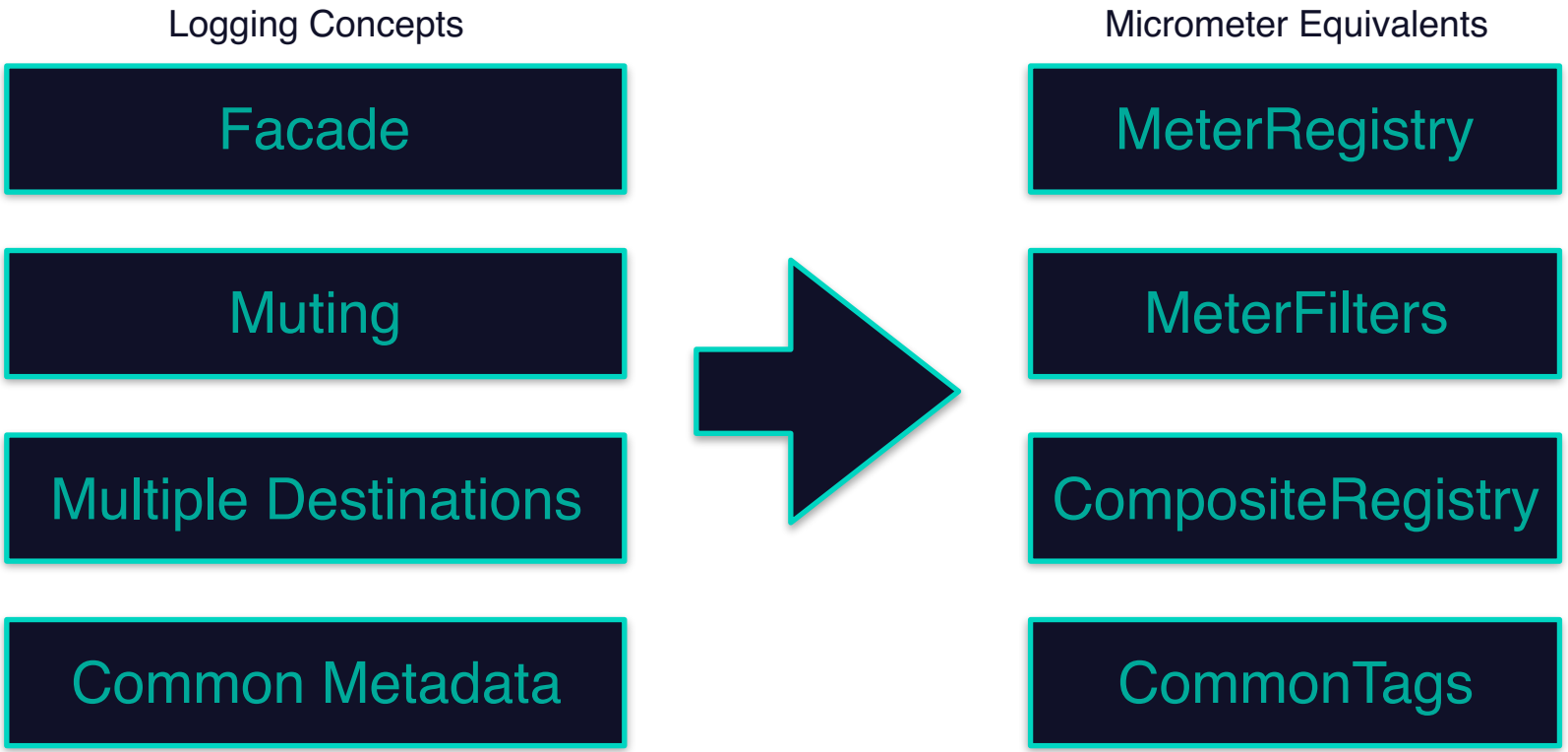
Observability Libraries:

1. Logging
    - SLF4J, Log4J, Logback, JUL, etc
2. Metrics
    - Micrometer, Prometheus, Drop Wizard Metrics, etc
3. Tracing
    - Zipkin

Key Logging Features

- Lots of libraries may need to log
  - They should use a logging facade like SLF4J
  - Shouldn't be tie users to a specific implementation
- Some log messages are very detailed and should allow muting
- The user may want to log to multiple destinations:
  - to console, to a file, and to a centralized logging system
- User may have cross cutting metadata they need to add to all messages

Micrometer Logging Similarities



Micrometer Terms

Meter - A measured ‘thing’

Examples: counters, timers, gauges, etc.

MeterRegistry - Meter store abstraction

Tag - A meter dimension

Metric - An individual measurement

Examples: Each timer by default creates 3 metrics: count, duration, max.

Demo #2

Micrometer without Spring
in Kotlin

Simple, Logging, Composite Meter Registry
MeterFilters
Counter, Timer, Gauge

Metric Cardinality (How many)

```
http_request 10
```

```
http_request{uri="users", method="GET"} 4
http_request{uri="user/{id}", method="GET"} 3
http_request{uri="user/{id}", method="PUT"} 3
```

```
http_request{uri="user/1", method="GET"} 1
http_request{uri="user/2", method="GET"} 1
http_request{uri="user/3", method="GET"} 1
http_request{uri="user/😳", method="GET"} 1
```

Cardinality Explosion

Rapid increase of metrics, typically due to storing a unique id or similar value as a tag

Consequences

- Increased memory usage
- Increased monitoring system load
- Increased monitoring system costs

How to keep tags under control

- Don’t use user input (directly)
- Use a MeterFilter to
  - Disable noisy meters
  - Rewrite high cardinality tags
  - Cap your total meter count
- Drop unwanted metrics at collection (Prometheus ‘relabeling’)

Spring Micrometer Integration

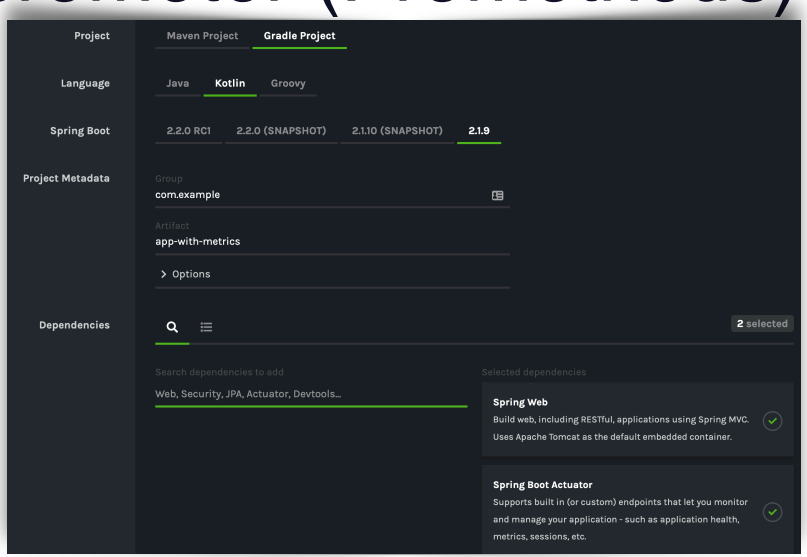
1. Built into Spring
2. Autowired by Spring
3. Integration provided by Micrometer
4. Integration provided by the library
5. Custom stuff!

Setting up Spring with Micrometer (Prometheus)

start.spring.io:

- Add ‘web’
- Add ‘actuator’)

Add Prometheus Registry (not on initializer)




```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-actuator")
    implementation("org.springframework.boot:spring-boot-starter-web")

    implementation("io.micrometer:micrometer-registry-prometheus:latest.release")
}
```

Enable Prometheus Actuator

```
management:
  endpoints:
    web:
      exposure:
        include: info,health,prometheus,metrics
```

‘Built in’ Metrics

System (File system, CPU, Uptime)

JVM (Heap, Class Loader, Garbage Collection)

HTTP Requests (Status, URI, Duration)

Tomcat Connections (Threads, Bytes Sent, Session, Errors)

Logging

```
sum_over_time(logback_events{level=“error”}[1h]) >
   sum_over_time(logback_events{level=“error”}[1h]) offset 1d * 1.5
```

The Metrics Actuator

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

Powered by Micrometer

Consider it a compatibility shim


```
{
   "name":"http.server.requests",
   "description":null,
   "baseUnit":"seconds",
   "measurements":[
      {
         "statistic":"COUNT",
         "value":22501.0
      },
      {
         "statistic":"TOTAL_TIME",
         "value":30048.789975875996
      },
      {
         "statistic":"MAX",
         "value":0.0
      }
   ],
   "availableTags":[
      {
         "tag":"method",
         "values":[
            "GET"
         ]
      },
```

Converting Health Checks To Metrics

- Not built in by default due custom ‘Health States’ being available.
- See https://micrometer.io/docs/guide/healthAsGauge
- Keep them fast since gauges are checked at metric collection time.

```
@FunctionalInterface
public interface HealthIndicator {
     Health health();
}

// healthIndicators: Map<String, HealthIndicator>
for ((key, value) in healthIndicators) {
  val tagKey = Tags.of("name", key)
  registry.gauge("health.indicator", tagKey,
this) {
    val status = value.health().status
    when (status.code) {
     "UP" -> 1.0
     "DOWN" -> -1.0
     "OUT_OF_SERVICE" -> -2.0
     "UNKNOWN" -> -3.0
     else -> -3.0
   }
  }
}
```

Add RestTemplate

Create via RestBuilder (To receive automatic Micrometer support see MetricsClientHttpRequestInterceptor)

```
private val restTemplate = restTemplateBuilder.build()
```

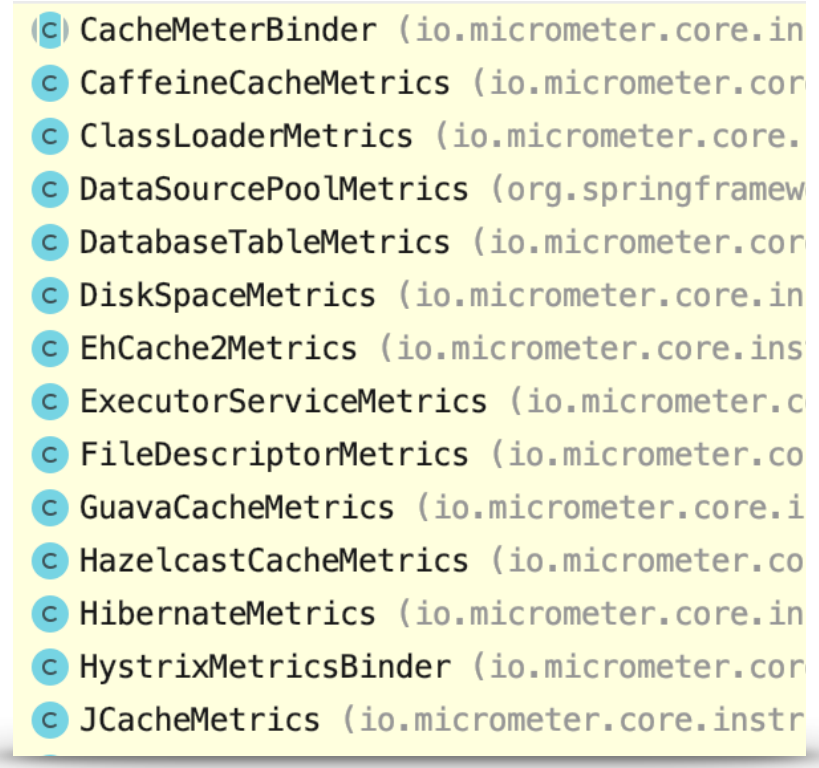
Use URL templating (Avoid Cardinality Explosion!)

```
private fun fetchUsers() : List<User>? {
    val shouldFail = Random.nextInt(1,5)
    return restTemplate.getForObject("http://localhost:8083/users/{shouldFail}", shouldFail)
}
```

Binder Interface

```
public interface MeterBinder {
    void bindTo(@NonNull MeterRegistry registry);
}
```

Allows adding metrics to MeterRegistry



Add Caching

Cache Manager Support (`@Cacheable`)

Many Cache Binders (Guava, Caffeine, EhCache, etc):

```
val userCache = CaffeineCacheMetrics.monitor(meterRegistry,
                    Caffeine.newBuilder().maximumSize(1).build<String, List<User>>(),
                    “users")
```

Add Resilience4J

Includes Micrometer support directly

Circuit Breaker State
Success/Failure rates
And much more!

Demo #3

Micrometer with Spring!

Binders

Config Properties

Percentiles

`actuator/metrics` and `actuator/prometheus`

Health Checks

Front End Metrics

No built in support for front end metrics

Potential for an actuator

Demo #4

Custom metrics

‘Browser’ metrics

More Micrometer!

Code examples at:
https://github.com/checketts/micrometer-springone-2019

Metrics for the Win: Using Micrometer to Understand Application Behavior

Wednesday
4:20pm–5:30pm

Erin Schnabel

Real-Time Performance Analysis of Data-Processing Pipelines with Spring Cloud Data Flow, Micrometer

Wednesday
4:20pm–5:30pm

Christian Tzolov and Sabby Anandan