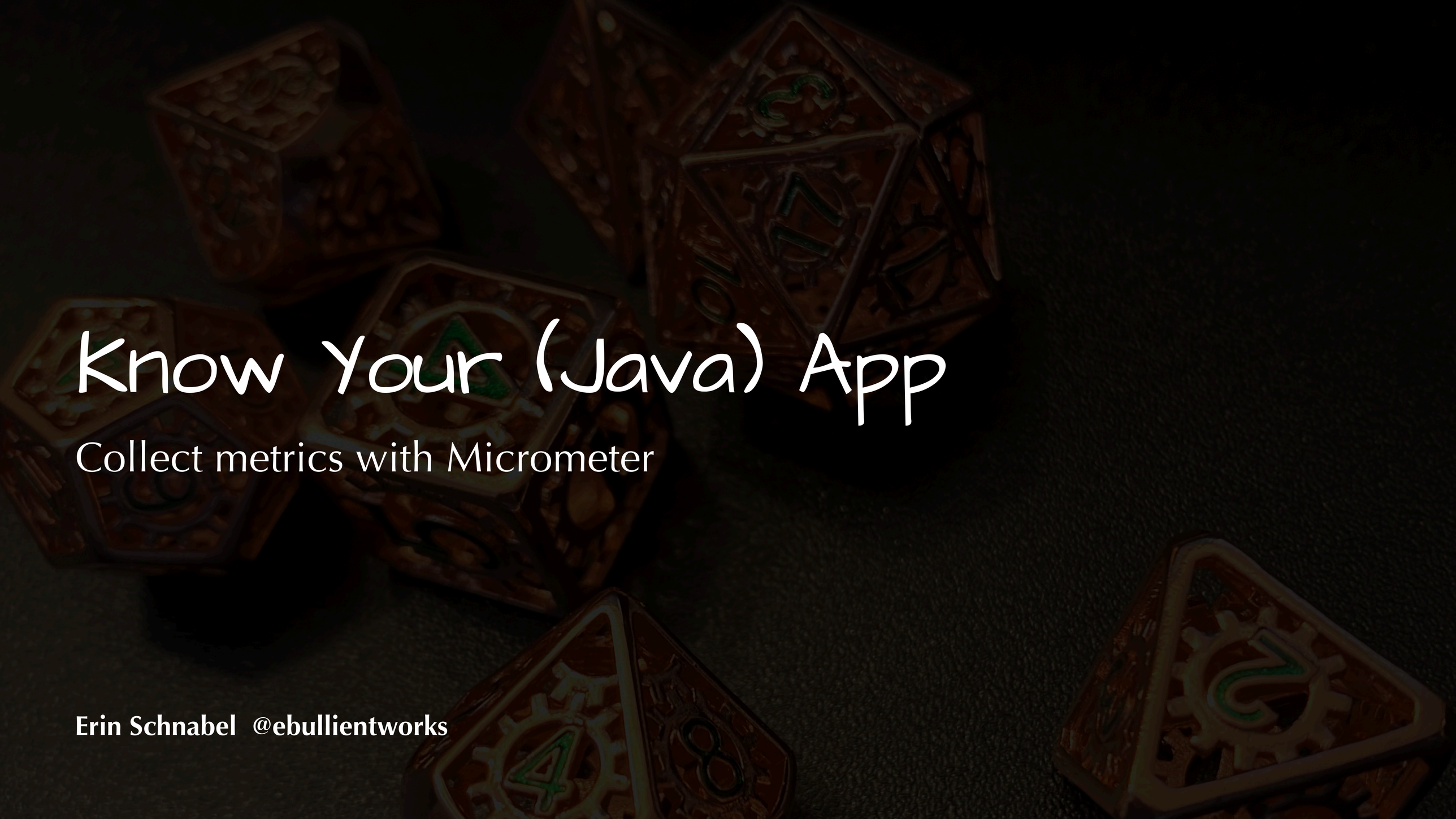
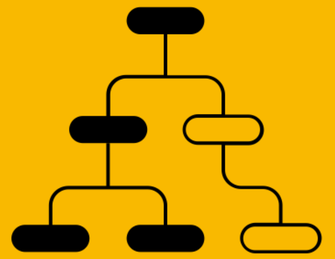


Know Your (Java) App

Collect metrics with Micrometer

Erin Schnabel @ebullientworks

What makes a system (or service) observable?

Health Checks	Metrics	Log Entries	Distributed Trace
Workload routing System health	Statistics & trends Analytics	Service-centric problem determination	Context + relationships for end–to-end analysis
			
Service is ready Service is not a zombie	How many times was method x called?	What happened when method x was called	Method x was called

Metrics: Time series data

How does data change *over time*?

- Time is the x-axis
  - Data collected at regular intervals
  - Each measured value appended to the end of a series

- Used for **trend analysis**, **anomaly detection**, **alerting**

Not for debugging, diagnostics, or root cause analysis

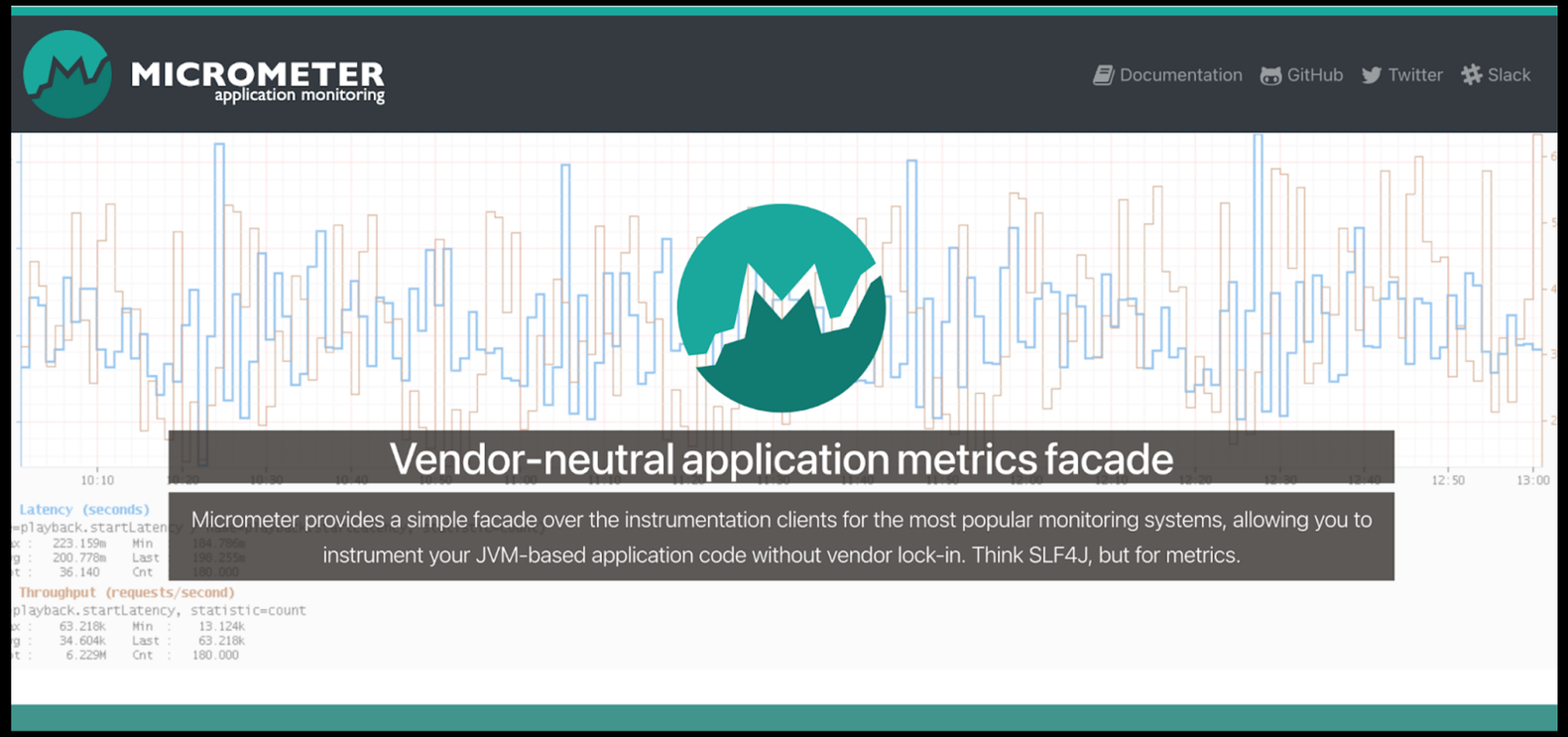
Metrics: Dimensions

Filtered/Selective aggregation

```
http_server_requests_seconds_count{
    env="test",
    method="GET",
    outcome="SUCCESS",
    status="200",
    uri="/example/prime/{number}",} 5.0
```

Cardinality: Each unique combination (name & all tag/values) is its own time series

What is Micrometer?



https://micrometer.io -- Open Source metrics library

"Screw gauge" micrometer

Developer API

- Obtain a `MeterRegistry`
- Create a `Meter` (name + tag=value combination)

`Gauge`
`Counter`
`Timer`
`DistributionSummary`
	`FunctionCounter`
	`FunctionTimer, LongTaskTimer`

Counter: increase only → .increment()

1. `registry.counter("example.prime.number", "type", "prime");`

2.
```
Counter.builder("counter")
       .baseUnit("beans")             // optional
       .description("a description") // optional
       .tags("region", "test")       // optional
       .register(registry);
```

3. `@Counted(value = "metric.all", extraTags = { "region", "test" })` **

Timer → .record(duration)

1. `registry.timer("my.timer", "region", "test");`

2.
```
Timer.builder("my.timer")
        .description("description ") // optional
        .tags("region", "test") // optional
        .register(registry);
```

3. `@Timed(value = "call", extraTags = {"region", "test"})` **

Working with Timers

1. `timer.record(() → noReturnValue());`
2. `timer.recordCallable(() → returnValue());`
3. `Runnable r = timer.wrap(() → noReturnValue());`
4. `Callable c = timer.wrap(() → returnValue());`
5. `Sample s = Timer.start(registry); doStuff; s.stop(timer);`

Gauge: increase/decrease, observed value

1.
```
List<String> list = registry.gauge("list.size", Tags.of( ... ),
        new ArrayList<>(), List::size);
```

2.
```
Gauge.builder("jvm.threads.peak", threadBean, ThreadMXBean::getPeakThreadCount)
        .tags("region", "test")
        .description("The peak live thread count ... ")
        .baseUnit(BaseUnits.THREADS)
        .register(registry);
```

FunctionCounter: gauge-like, counter function

1. `registry.more().counter( ... )`

2.
```
FunctionCounter.builder("hibernate.transactions", statistics,
        s -> s.getTransactionCount() - s.getSuccessfulTransactionCount())
          .tags("entityManagerFactory", sessionFactoryName)
          .tags("result", "failure")
          .description("The number of transactions we know to have failed")
          .register(registry);
```

FunctionTimer: gauge-like, counter & duration function

1. `registry.more().timer( ... );`

2.
```
FunctionTimer.builder("cache.gets.latency", cache,
        c → c.getLocalMapStats().getGetOperationCount(),
        c → c.getLocalMapStats().getTotalGetLatency(),
        TimeUnit.NANOSECONDS)
          .tags("name", cache.getName())
          .description("Cache gets")
          .register(registry);
```

LongTaskTimer (active count, longest active task)

1. `registry.more().longTaskTimer("long.task", "region", "test");`

2.
```
LongTaskTimer.builder("long.task")
        .description("a description") // optional
        .tags("region", "test") // optional
        .register(registry);
```

3. `@Timed(value = "long.task", extraTags = {"extra", "tag"}, longTask = true)` **

Distribution Summary → .record(value), not time

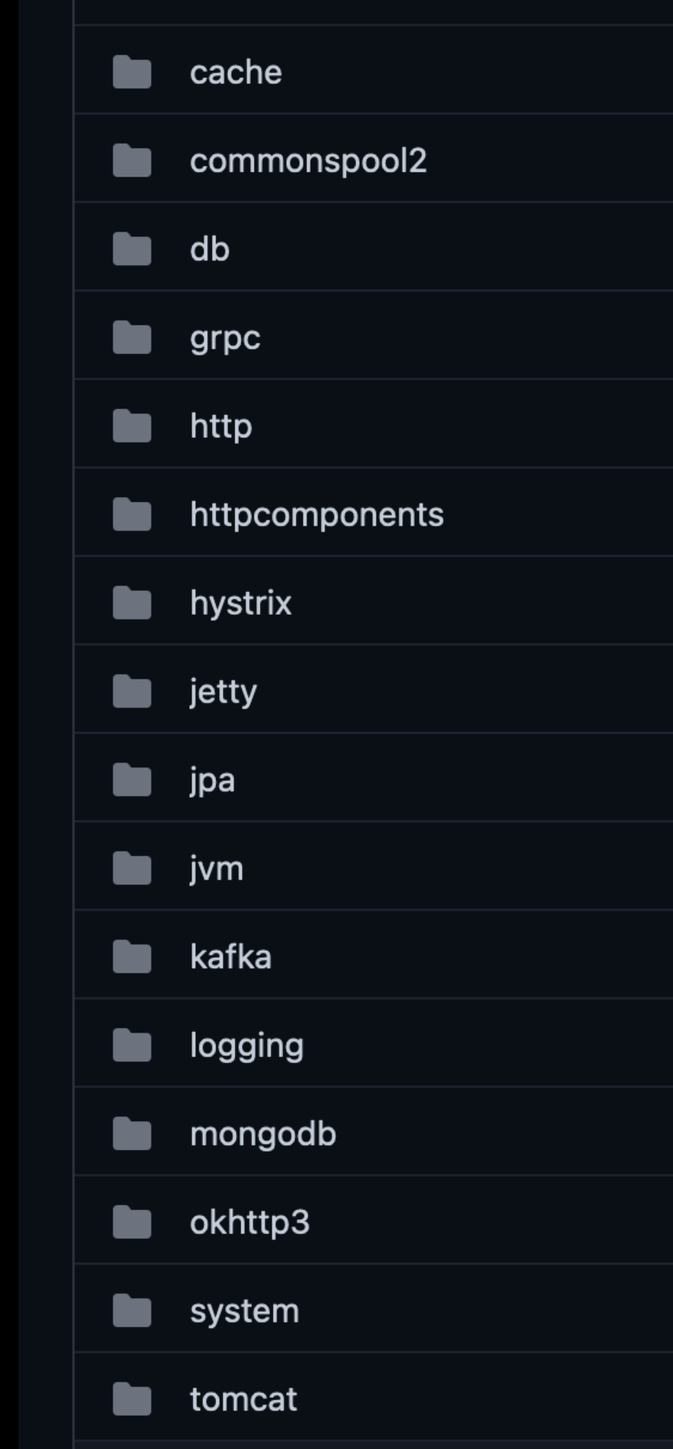
1. `registry.summary("name", "region", "test")`

2.
```
DistributionSummary.builder("response.size")
    .description("a description") // optional
    .baseUnit("bytes")            // optional
    .tags("region", "test")       // optional
    .register(registry);
```

Library Maintainer API

- Provide a `MeterBinder` (optional dependency)
  - Register library-specific metrics
- Quarkus / Spring discover `MeterBinder` implementations

```
binder.bindTo(registry)
```



Configuration and Filtering

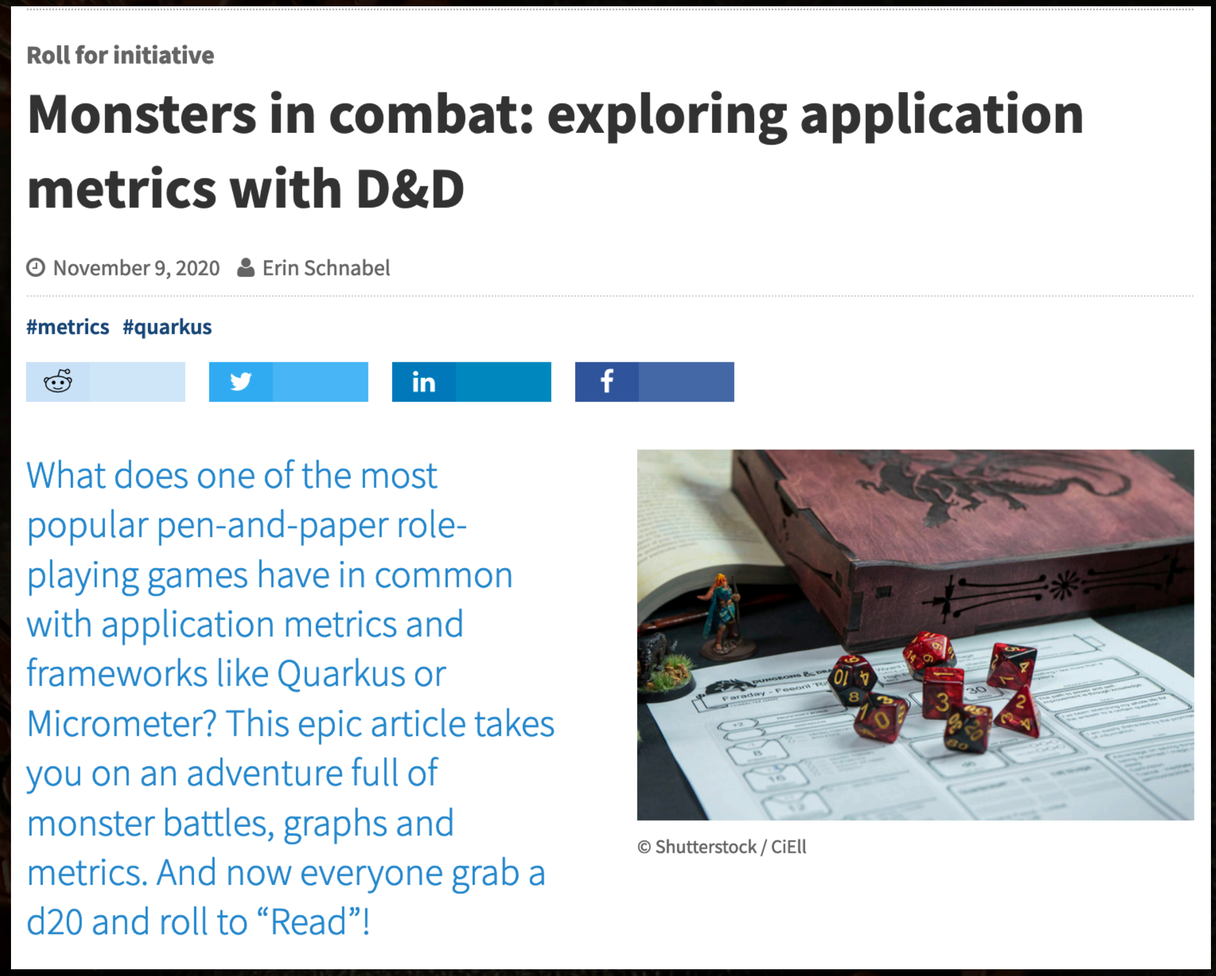
- Late-binding configuration provided by `MeterFilter`
  - Sourced from user code or shared library

- Accept/Deny filters: fine-grained control over gathered metrics
- Transform: rename metrics, add/ignore/replace/rename tags
- Customize Timer / Histogram behavior
  - Percentiles, quantiles, Service Level Agreement(SLA) boundaries

Monster Combat

https://github.com/ebullient/monster-combat

https://twitter.com/ebullientworks/status/1328835279659085828?s=20



https://jaxenter.com/metrics-dnd-173311.html

Rolling dice

Micrometer:

```
Dice.setMonitor((k, v) -> {
    registry.counter("dice.rolls", "die", k, "face", label(v))
            .increment();
});
```

Attacks : Hits and Misses

- AC == Armor class

  Attacker rolls a d20...

  20 – critical hit (HIT)
  1 – critical miss (MISS)

- DC == Difficulty class

  Defender rolls a d20...

```
: oneRound:
   Troll(LARGE GIANT){AC:15,HP:84(8d10+40),STR:18(+4),DEX:13(+1),CON:20(+5),INT:7(-2),WIS:9(-1),C
   Pit Fiend(LARGE FIEND){AC:19,HP:300(24d10+168),STR:26(+8),DEX:14(+2),CON:24(+7),INT:22(+6),W

: attack: miss: Troll(36) -> Pit Fiend(100)
: attack: miss: Troll(36) -> Pit Fiend(100)
: attack: hit> Troll(36) -> Pit Fiend(97) for 9 damage using Claws[7hit,11(2d6+4)|slashing]
: attack: hit> Pit Fiend(97) -> Troll(10) for 22 damage using Bite[14hit,22(4d6+8)|piercing]
: attack: MISS: Pit Fiend(97) -> Troll(10)
: attack: HIT> Pit Fiend(97) -> Troll(0) for 34 damage using Mace[14hit,15(2d6+8)|bludgeoning]

: oneRound: survivors
   Pit Fiend(LARGE FIEND){AC:19,HP:300(24d10+168),STR:26(+8),DEX:14(+2),CON:24(+7),INT:22(+6),W
```

Micrometer: Timer & Distribution Summary

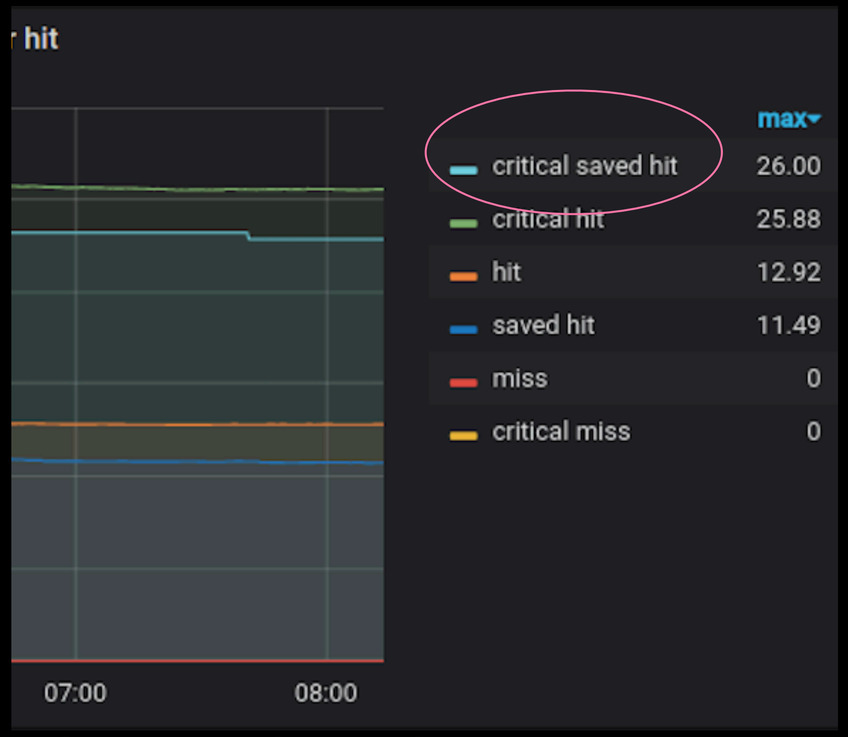
- Records value

```
registry.summary("round.attacks",
        "hitOrMiss", event.hitOrMiss(),
        "attackType", event.getAttackType(),
        "damageType", event.getType())
        .record((double) event.getDamageAmount());
```

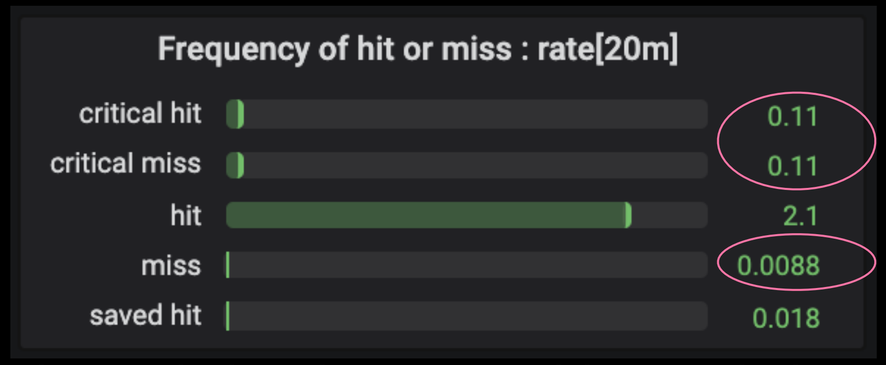
- Default metrics: count, sum, max
- Optional: cumulative histogram buckets, pre-computed quantile

So many bugs...

There is often no practical way to write tests for all permutations...



Who wrote this code, anyway?



This ended up being a problem with the original conversion from HTML to JSON.

Questions?