



Observable Microservices

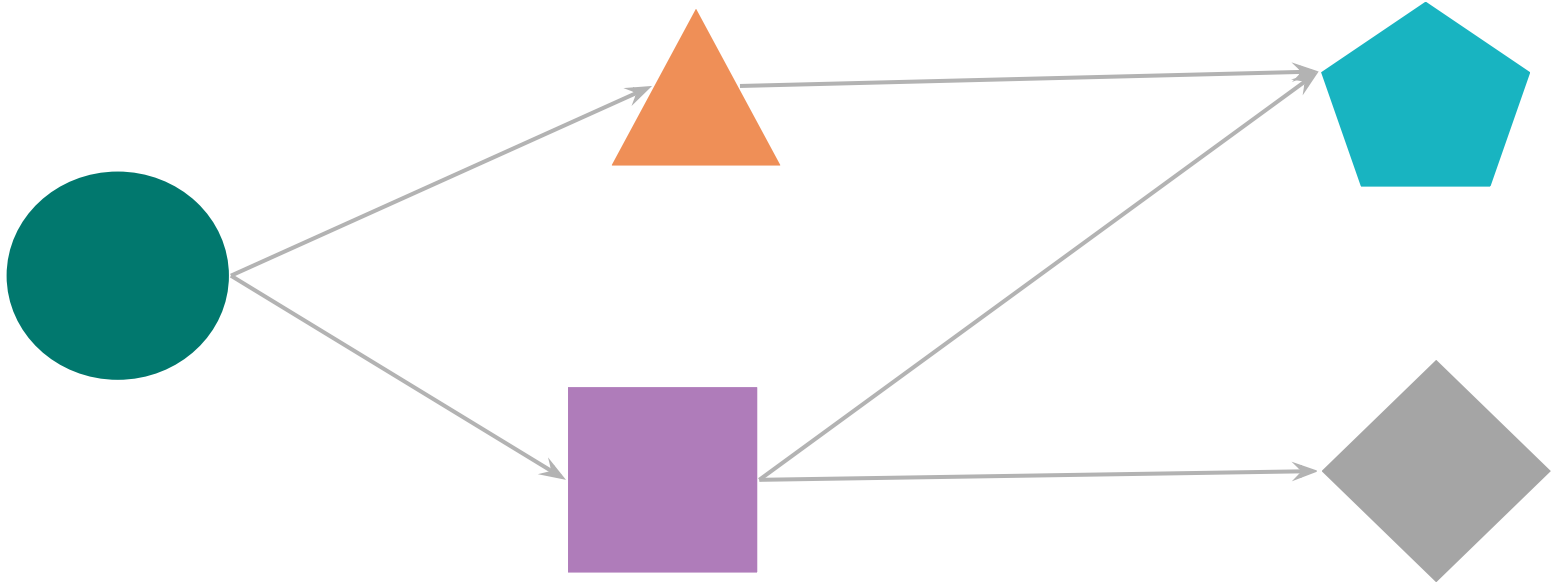
Adib Saikali - @asaikali
asaikali@pivotal.io

Go to: <https://github.com/asaikali/boot-observability>

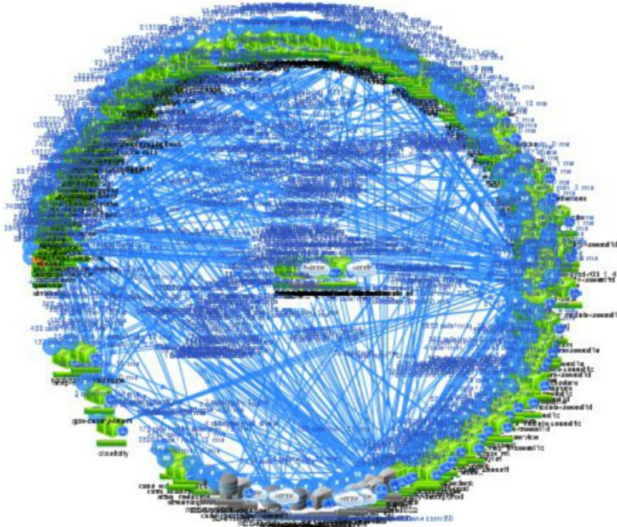
Execute: git pull or clone <https://github.com/asaikali/boot-observability.git>

Execute: ./mvnw clean package

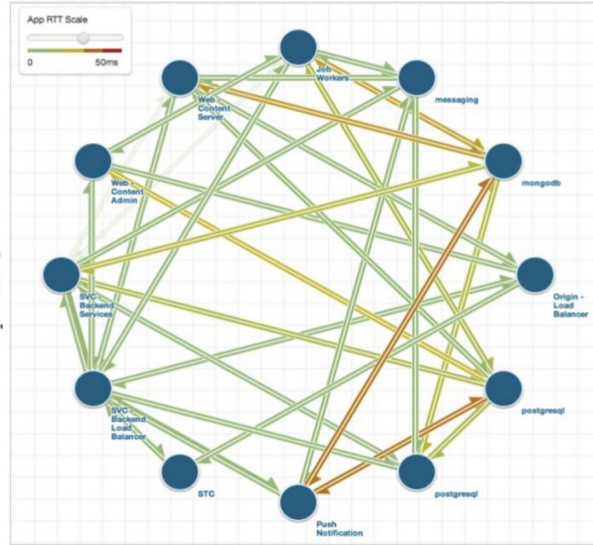
Microservices Compose Other Microservices



Death Star Architecture



Netflix



Gilt Groupe (12 of 450)



Twitter

As visualized by Appdynamics, Boundary.com and Twitter internal tools

Observability According to Wikipedia

“In control theory, **observability** is a **measure of how well internal states** of a system **can be inferred** from knowledge of its external outputs”

<https://en.wikipedia.org/wiki/Observability>

Four Types Of Observability

- Logging
- Healthchecks
- Metrics
- Distributed Tracing

Observability



Logging

Traditional Unstructured Logging

```

      .      ____          _            __ _ _
/\ \    / __ _      / \   / \   / \   / \   \ \
(  )  / |`'_ |     / _ | / _ | / _ | / _ |  ) )
\ \ /  | |_) |   / __|/ __|/ __|/ __|/ __|/ __|
  ( /   | |_) | /  __/|  __/|  __/|  __/|  __/
   \_\   \____|/  __/|  __/|  __/|  __/|  __/
      |_____|____/|____/|____/|____/|____/

=====|_|=====|____/____/____/____/

:: Spring Boot ::      (v2.2.4.RELEASE)

2020-02-18 22:13:15.326 INFO 19548 --- [main] com.example.MetricsApplication : Starting MetricsApplication on adib.local with PID 19548 (/Users/adib/Projects/learn-spring-boot-2020/target/classes)
2020-02-18 22:13:15.329 INFO 19548 --- [main] com.example.MetricsApplication : No active profile set, falling back to default profiles: default
2020-02-18 22:13:16.448 INFO 19548 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-02-18 22:13:16.455 INFO 19548 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-02-18 22:13:16.455 INFO 19548 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.30]
2020-02-18 22:13:16.516 INFO 19548 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-02-18 22:13:16.517 INFO 19548 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1139 ms
2020-02-18 22:13:16.859 INFO 19548 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-02-18 22:13:17.048 INFO 19548 --- [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 14 endpoint(s) beneath base path '/actuator'
2020-02-18 22:13:17.116 INFO 19548 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2020-02-18 22:13:17.119 INFO 19548 --- [main] com.example.MetricsApplication : Started MetricsApplication in 2.085 seconds (JVM running for 2.592)
2020-02-18 22:13:17.195 INFO 19548 --- [1]-172.20.0.142] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-02-18 22:13:17.195 INFO 19548 --- [1]-172.20.0.142] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-02-18 22:13:17.201 INFO 19548 --- [1]-172.20.0.142] o.s.web.servlet.DispatcherServlet : Completed initialization in 6 ms

```

- Unstructured logs are hard to query and search through, since there is no standard format for the file
- Not possible to tell from the log statements what version of the code wrote the statement

Structured Logging

```
{
  "@timestamp": "2020-02-18T22:23:12.361-05:00",
  "@version": "1",
  "message": "Hello called foo=123 -> world",
  "logger_name": "com.example.HelloController",
  "thread_name": "http-nio-8080-exec-3",
  "level": "INFO",
  "level_value": 20000,
  "foo": 123,
  "bar": "world",
  "commitId": "35aba34001dad0cb5fcbfe52233d7e7c5c6f36af"
}
```

- Structured logs are written using a well know format such as JSON that is easy to index
- Structured logs can be easily queried
- Adding the git commit id to every log event makes it possible to determine precisely which version of the code wrote the log message

logs

- Run the structured logging application
- Observe the output of the logs
- Visit the application at `localhost:8080`
- Check the output written to the log
- Examine the `HelloController` and notice how the extra fields are written to the log

Observability



Health Checks

Health checks

- Run the `HealthApplication`
- Visit localhost:8080/actuator/health
- Examine the output
- Visit localhost:8080/fail
- Visit localhost:8080/actuator/health
- Visit localhost:8080/pass
- Visit localhost:8080/actuator/health
- Examine the code in `ExampleHealthIndicator`

Health check Groups

- Visit localhost:8080/actuator/health/foo
- Visit localhost:8080/actuator/health/bar
- Examine `application.yml`

Observability



Distributed Tracing

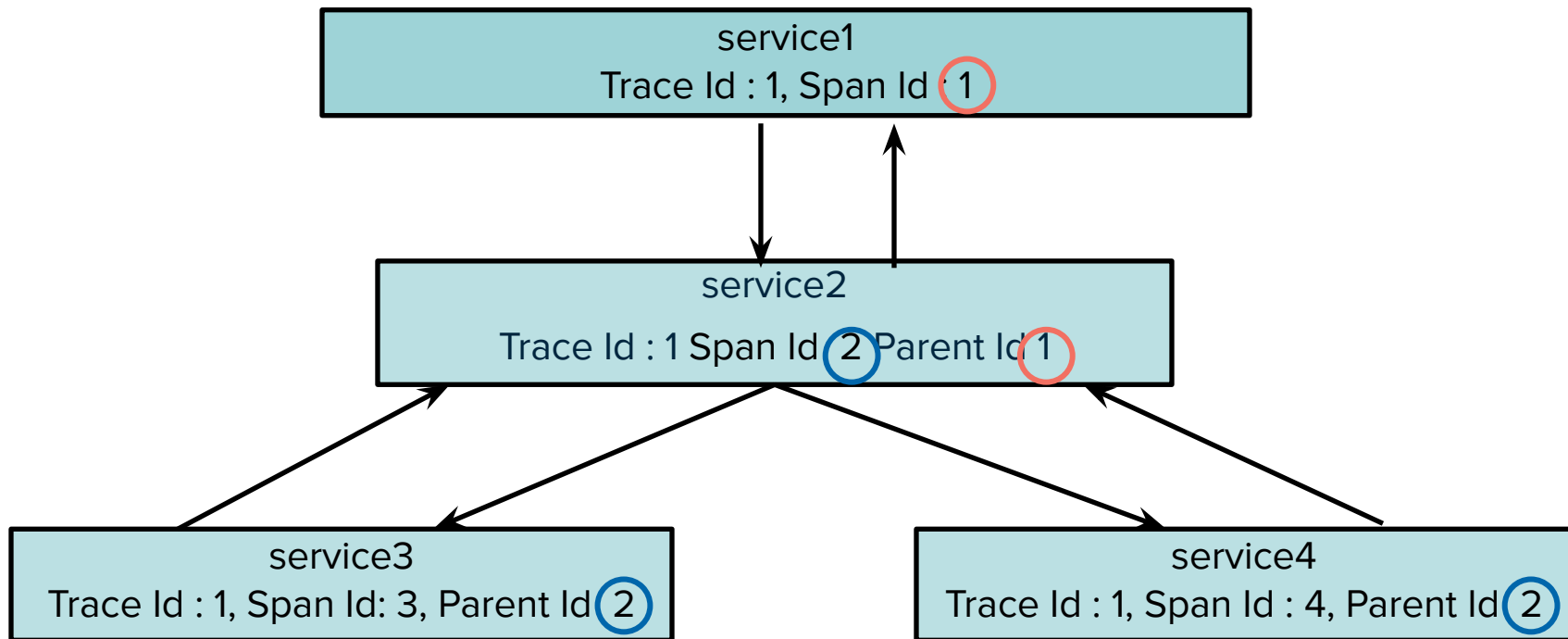
Troubleshooting Latency Issues

- When was the event? How long did it take?
- How do I know it was slow?
- Why did it take so long?
- Which microservice was responsible?

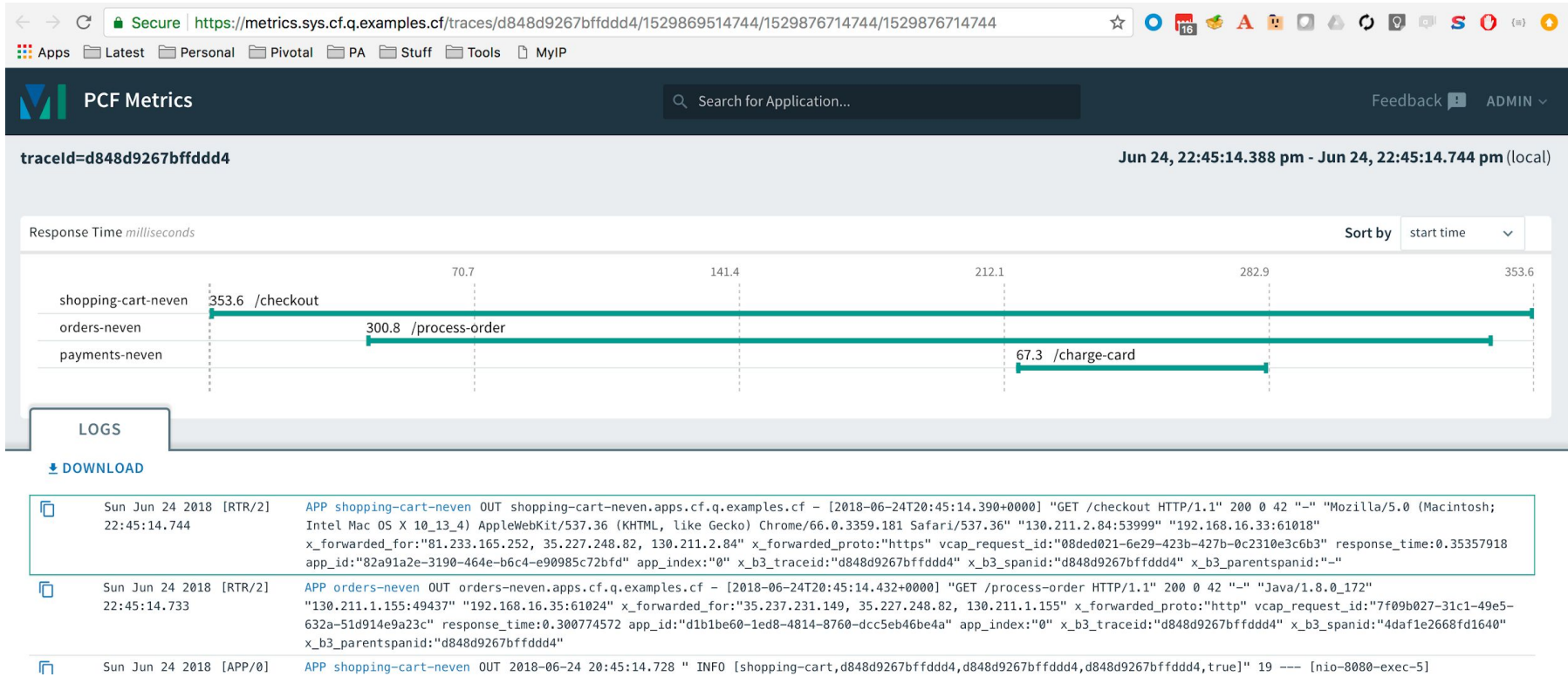
Distributed Tracing

- **Distributed Tracing** is a process of collecting end-to-end transaction graphs in near real time
- A **trace** represents the entire journey of a request
- A **span** represents single operation call
- Distributed Tracing systems are often used for this purpose.
Zipkin is an example
- **Tracers** add logic to create unique trace Id, span Id

Visualization - Traces and Spans



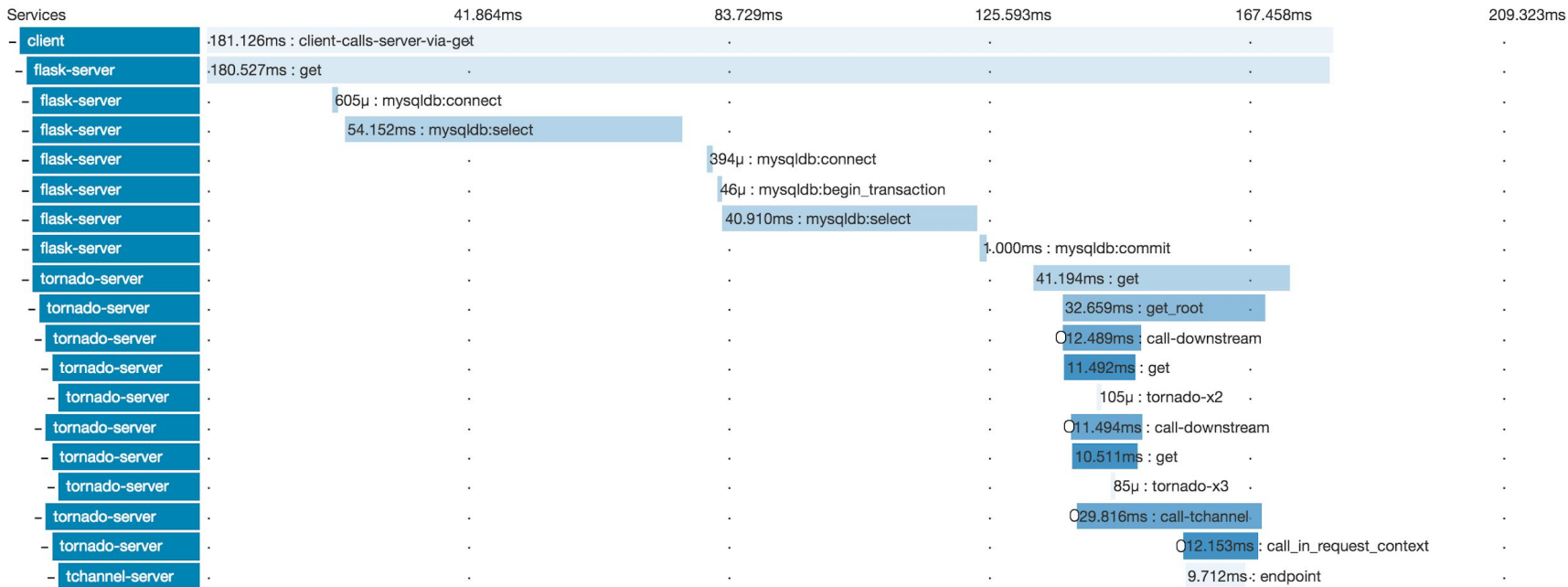
PCF Metrics



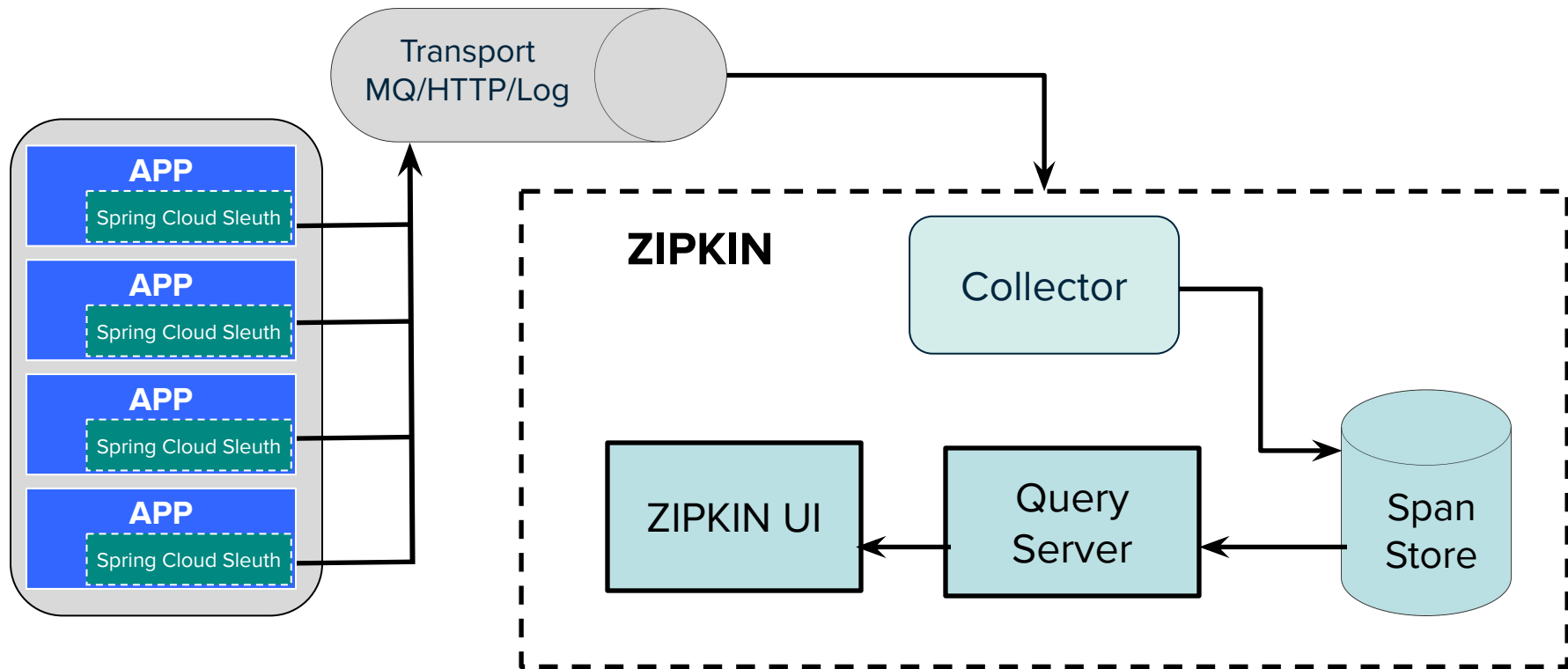
Duration: **209.323ms**
Services: **5**
Depth: **7**
Total Spans: **24**
[JSON](#)

Expand All
Collapse All
Filter Service Se...

client x4
flask-server x10
missing-service-name x2
tchannel-server x2
tornado-server x11



Integrate distributed tracing to your apps



Spring Cloud Sleuth

<https://github.com/practical-microservices/spring-cloud-sleuth-basics>

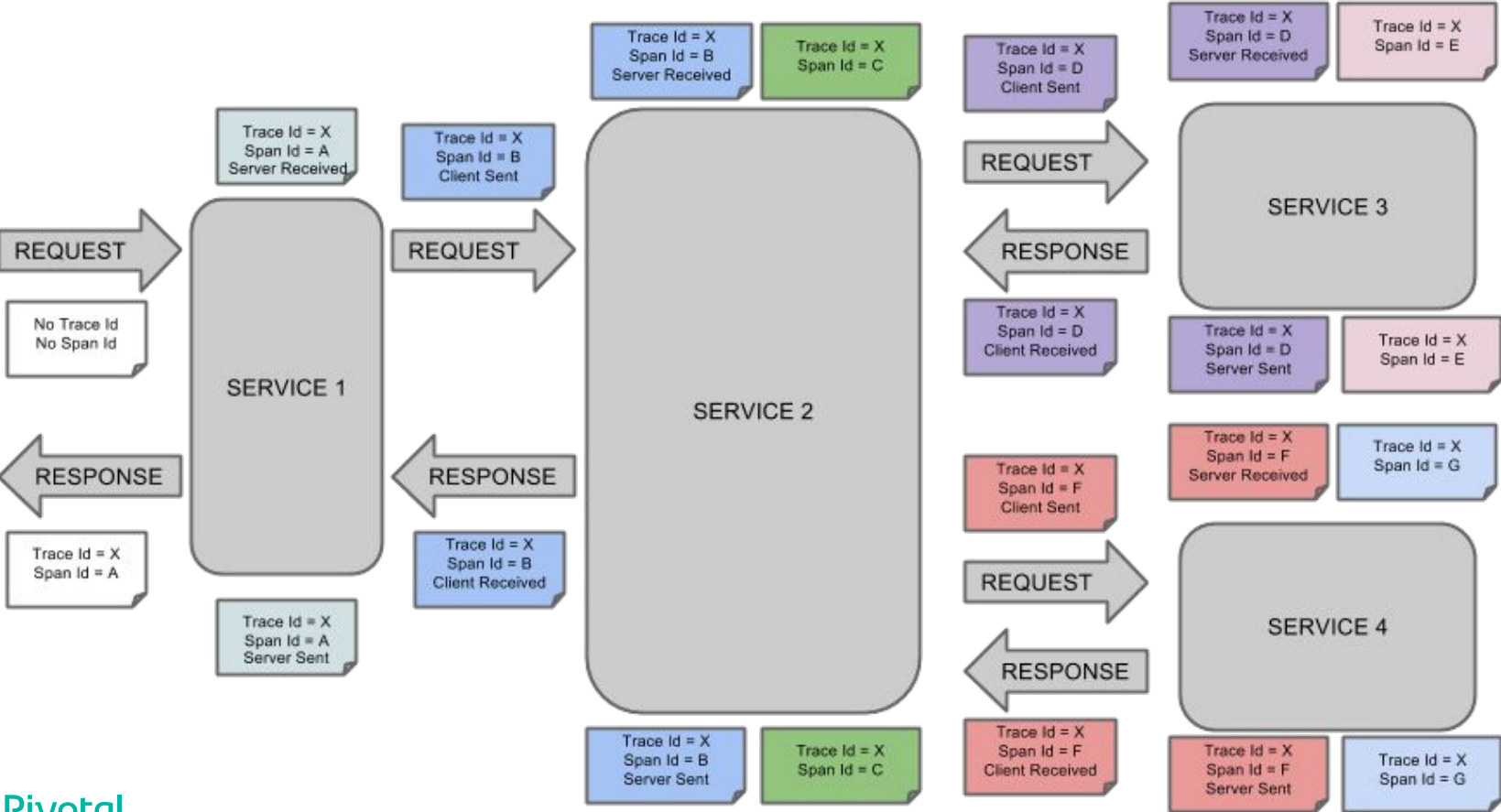
PROJECTS : SPRING CLOUD

Spring Cloud Sleuth

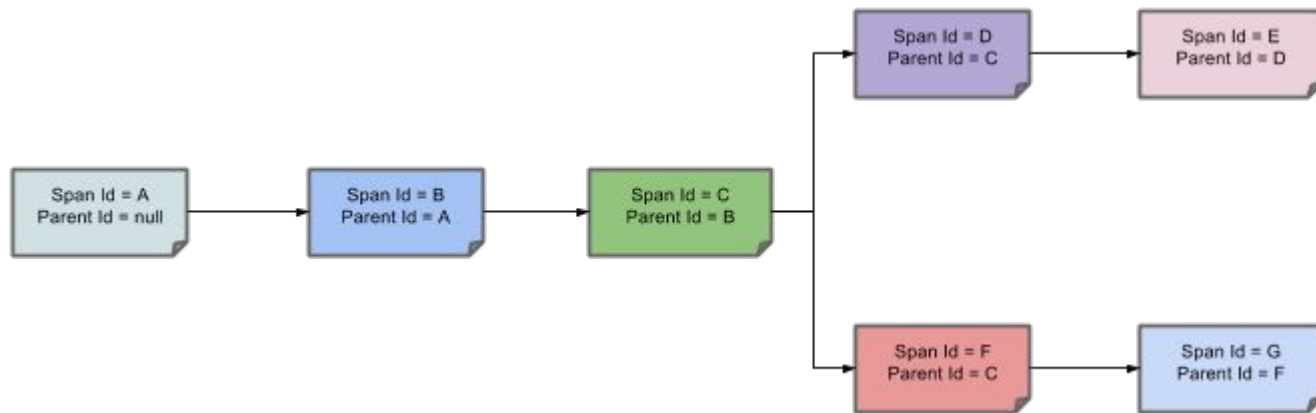


Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud, borrowing heavily from [Dapper](#), [Zipkin](#) and HTrace. For most users Sleuth should be invisible, and all your interactions with external systems should be instrumented automatically. You can capture data simply in logs, or by sending it to a remote collector service.

Spring Cloud Sleuth



Tree built from trace data



Distributed Tracing

- Run `MessageService` in `tracing/message-service` project
- Run `BillboardClientApplication` in `tracing/billboard-client` project
- Open a terminal
 - `cd` into `tools/zipkin`
 - Run `java -jar zipkin-server-2.19.3-exec.jar`
 - Visit localhost:9411
- Visit localhost:8080 wait for the quotes to start loading
- Go back to zipkin ui
 - Search for traces to find a trace
 - Inspect the trace details
 - Show the dependency map

Observability



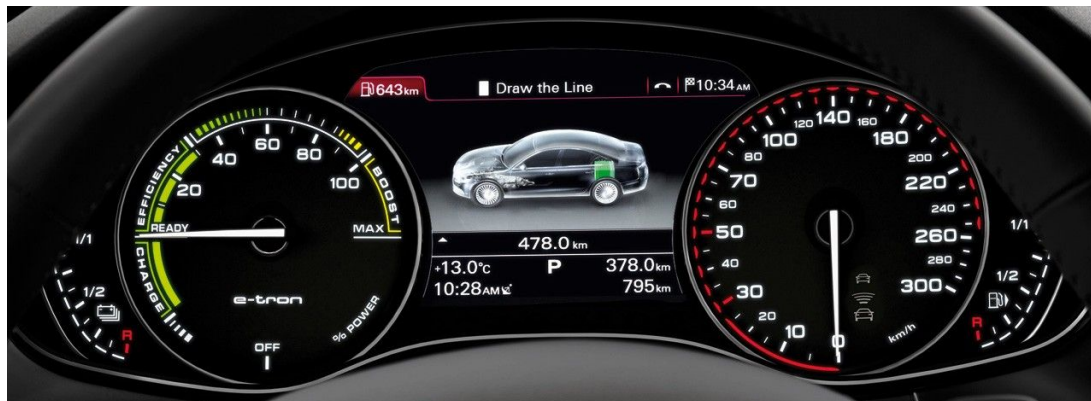
Metrics

The Problem Context

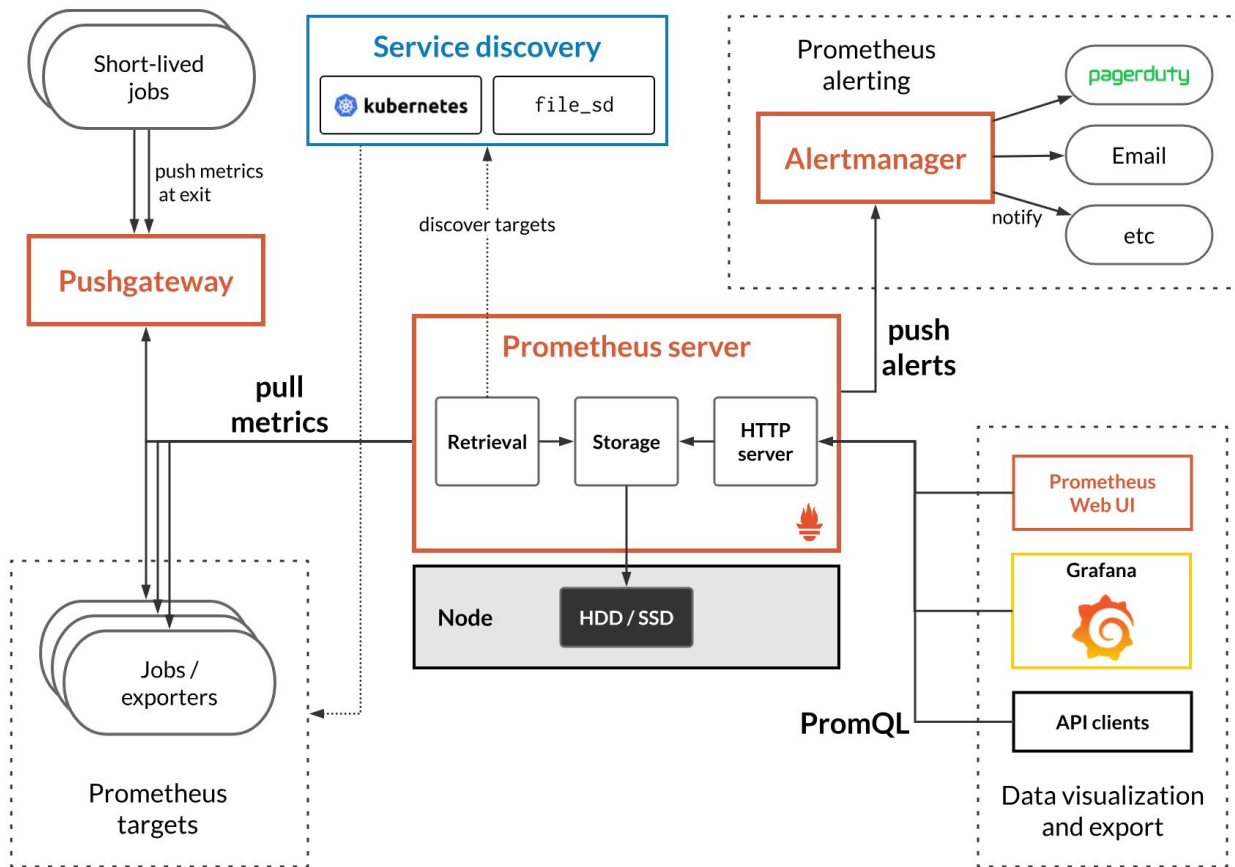
- How can a microservice expose metrics on its internal state?
- How can metrics be tagged to make analysis and dashboarding easy?
- How can metrics be made to follow the conventions of the monitoring system used to store the metrics?
- How can metrics be altered on?

Common Metric Types

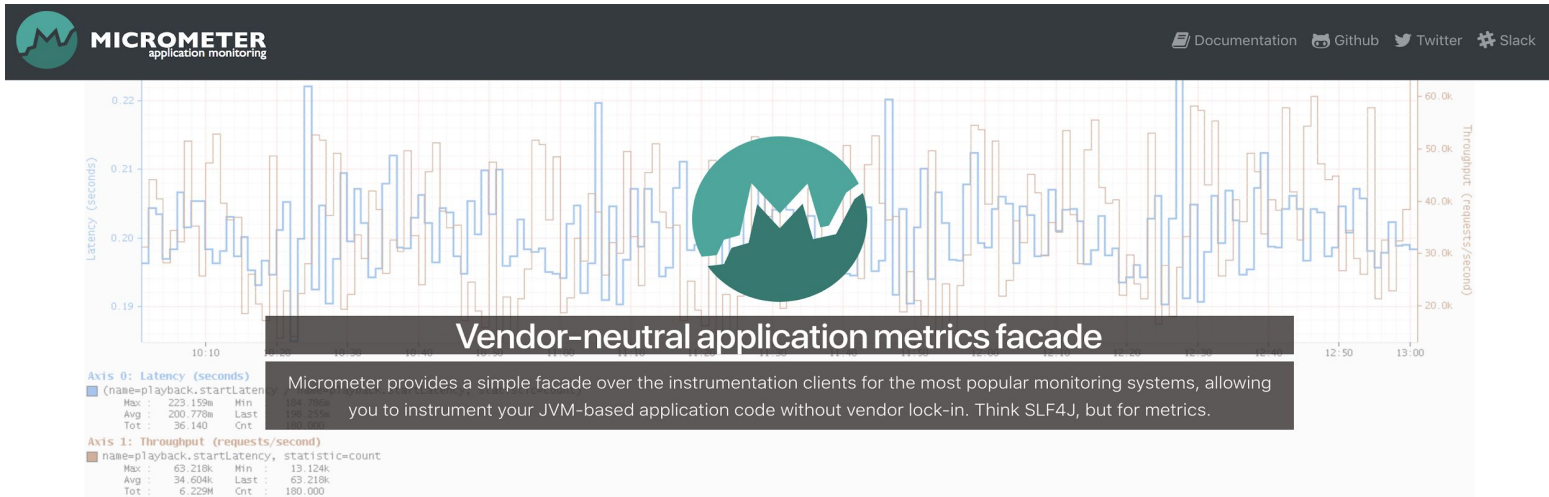
- **Counter** - always going up forever until reset back to 0
- **Gauge** - current value of a measure that range over a min and a max
- **Timer** - measures how long something took
- **Event Distributions & Histograms**



prometheus.io



Micrometer.io



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.



Pre-configured Bindings

Out-of-the-box instrumentation of caches, the class loader, garbage collection, processor utilization, thread pools, and more tailored to actionable insight.



Integrated into Spring

Starting with Spring Boot 2.0, Micrometer is the instrumentation library powering the delivery of application metrics from Spring. Support is ported back to Boot 1.x through an additional library dependency.

Metrics

- Launch Prometheus
 - Open a command shell
 - cd into tools/<YourOS>/prometheus
 - Run the prometheus binary
 - Visit localhost:9090
- Launch Sample Application
 - Run `MetricsApplication` in metrics project
 - Visit localhost:8080 a to generate some random metrics
 - Visit localhost:8080/actuator/prometheus to view the the prometheus metris emitted by the application via micrometer
- Go back the Prometheus UI and check for metrics that the app is emitting you might need to generate some more data to get values that can be graphed.
- Examine the code in metrics application



Pivotal®

Transforming How The World Builds Software