Leading Practices for Microservice Logging

Objectives

Key objectives of this chapter

- The Challenges of Logging in a Service Oriented Architecture
- A list of leading practices recommended for handling Microservice logging
- A discussion of those practices

1.1 Logging Challenges

- (Micro-)Service Architectures mean that solutions are distributed across a network topology, with many small processes involved in handling a single request.
- Simple logging solutions generally don't scale beyond a single process.
- How do we manage logs, which contain forensic information, when the requests have been distributed to many, potentially load-balanced, processes?

1.2 Leading Practices

- Correlate Requests with a Unique ID
- Include a Unique ID in the Response
- Send Logs to a Central Location
- Structure Your Log Data
- Add Context to Every Request
- Write Logs to Local Storage

Notes

We refer to these as leading practices because the notion of "best" is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: https://dzone.com/articles/microservices-logging-best-practices

1.3 Correlate Requests with a Unique ID

- Requests will flow through various instances of our microservices.
- By attaching, propagating, and using a unique ID for each request, we can identify the log records associated with handling that request.
- Since each request represents a call between two services, when logging requests, we could use a compound key consisting of a unique ID for the main (originating) request, as well as source and target IDs of each service. That way we would be able to find all entries related to the main request, as well as all requests related to a specific source or target service.

1.4 Include a Unique ID in the Response

- When providing a response, whether a fault or a regular response, provide a Unique ID.
  - The ID is typically the one used to correlate the request.
- In the event that you need more detail about a response, the Unique ID can be used to extract information from the logging system.
  - Imagine extracting the log records for a particular loan application to im working on itiminvestigate how the system determined its response.

1.5 Send Logs to a Central Location

- Microservice solutions are distributed across a network topology. If we do not collect log information in a central location, then when we have the need to analyze the logs, it will be extremely challenging to locate and gather all of the log records from wherever they may be in the network.
- To make matters even worse, if you log to local storage within a container, and that container is terminated, the log data may be lost entirely.
- For these reasons, centralized logging is now considered to be the norm, and a "solved problem" with well-known, packaged solutions.

1.6 Structure Your Log Data

- Microservice solutions tend to be extremely heterogeneous.
  - There may be no agreement on the content of the log records. Different services need different fields, and there is no point in having a bloat of unnecessary information.
  - Different technology stacks may used different raw logging formats and different mechanisms.
- Put your log information into a flexible, extensible, format such as JSON or XML.
  - Depending on your technology stack, you might do this natively, or via a logging sidecar service.

Notes

You may already be familiar with the notion of a sidecar, but if not, it will be discussed a bit later in this unit.

1.7 Add Context to Every Record

- The purpose for logging is to provide a record suitable for analysis and action.
- There are different audiences for the log.
  - Operations staff need complete, understandable, actionable, information on which to act in the event of issues. This allows for the timely remediation.
  - Developers may need more detailed information with which to debug. This level of detail may require knowledge of the source code in order to interpret.
  - Tools – we may be able to automate actions in response to certain log information, allowing seamless and rapid response without manual intervention.
- When logging, try to provide all of the context that any party may need in the future.
  - In general, include whatever information you believe would be, or later discover would have been, useful to resolving an issue.
  - Constantly evaluate the contents of your log records. After all, we choose to use a flexible and extensible format for reasons, including not being locked into a single set of content and allowing each audience segment to extract the log information relevant to it.

Notes

We refer to these as leading practices because the notion of "best" is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: https://dzone.com/articles/microservices-logging-best-practices

1.8 Examples of Content

- Examples of useful log content include:
  - Timestamp – every log record should have a timestamp, and your network should be time synchronized
  - Exception trace – log records related to exceptions should include their detailed stack traces
  - The service name and location – each log record should identify the service instance that generated it.
  - A code locator – indicates from where in the code the log record originated.
  - API level information – the operation, parameters, return values, status code, etc. involved.
  - Network addresses – source and target IP addresses
  - Client information – identify the app and/or user-agent that generated the request.
- Again, evaluate log content based on your use. What information are you logging that is useful? What are you are not logging that would have been useful? What information might help automate handling?

Notes

We refer to these as leading practices because the notion of "best" is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: https://dzone.com/articles/microservices-logging-best-practices

1.9 Write Logs to Local Storage

- Centralized logs are vital, but writing to them directly would be ill-advised. If you think of centralized logs as a service, you realize that writing to them is subject to the same issues as service to service communications.
- One approach is to write your logs locally, then use a sidecar (q.v., Service Mesh pattern) to transfer the local logs to the central logging service.
  - The sidecar can take care of transforming and transferring the local logs to the central logs in a timely, but efficient, manner. It also helps to decouple stack specific log formats from the canonical log format chosen for the central logging service.

⚠ When writing logs to local storage, be mindful of how logs are rotated and retained so they don't fill up the disk

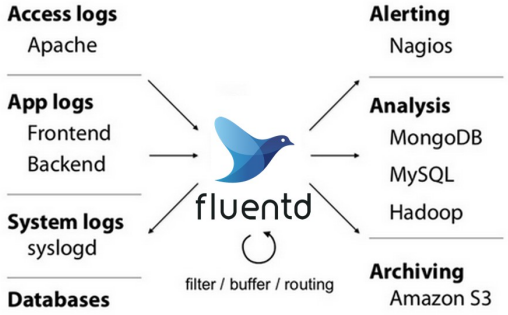
Notes

We refer to these as leading practices because the notion of "best" is a value judgment left to the implementer. These are generally considered the right practices with which to guide, or lead, adopters.

Ref: https://dzone.com/articles/microservices-logging-best-practices

1.10 Collecting Logs with Fluentd

- Fluentd is an open source data collector, which lets you unify the data collection and consumption for a better use and understanding of data.



- It's key features are:
  - Unified Logging with JSON
  - Pluggable Architecture (plugins both for sources and data outputs)
  - Minimum Resources Required (The vanilla instance runs on 30-40MB of memory and can process 13,000 events/second/core. If you have tighter memory requirements (-450kb)
  - Built-in Reliability (memory- and file-based buffering)

1.11 Leading Practices for Microservice Logging Summary

- We considered some of the challenges related to logging when your solutions are distributed and heterogeneous.
- We looked at a list of practices that can help you solve the logging challenges
- We discussed those practices.

1.12 Metrics Using Prometheus

- Overview
- Prometheus
- Service Discovery
- Exposing Metrics in Services
- Querying in Prometheus
- Grafana

1.13 Overview

- The idea of maintaining performance metrics is an old and well-understood concept.
  - For Java, consider JSR 174: Monitoring and Management Specification
- The challenge in modern service-oriented architectures is that our solutions are fine-grained and distributed across network topologies. Metrics have to be gathered, correlated and organized.

Notes

JSR 174: https://jcp.org/en/jsr/detail?id=174

1.14 Prometheus

- Prometheus is an Open Source tool, originally developed by SoundCloud, for collecting, storing and analyzing metrics in a network environment.
- Prometheus gather metrics from many sources throughout your network, stores them as time-series data, supports queries, and generates alerts.

1.15 Prometheus

- Major features provided by Prometheus include:
- A time-series database
- A purpose-built query language
- A web UI
- Support for third party tools, e.g.
  - Grafana – a high-end, Open Source, solution for analyzing and visualizing time-series data
  - Tools, such as prophet, forecast-prometheus or prometheus-anomaly-detector, which use various techniques to attempt to predict future failures based on patterns of change in metrics

Notes

Prometheus stores metrics in a time-series data model, q.v., https://en.wikipedia.org/wiki/Time_series

1.16 Prometheus

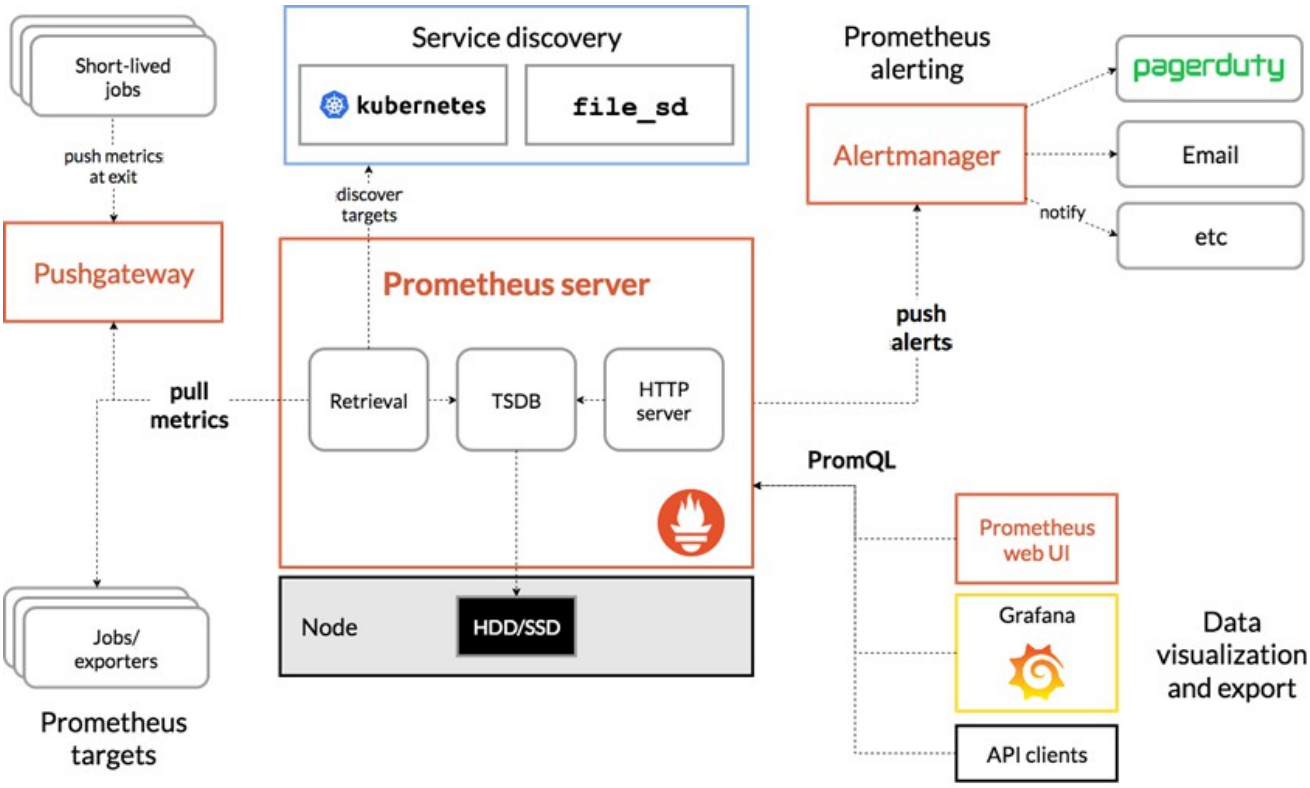
- Major components that make up Prometheus include:
  - A central server collecting metrics and hosting the time-series metrics database
  - Client libraries for instrumenting applications to expose metrics
  - Application/technology specific exporters that expose the metrics of existing, third-party, systems into Prometheus without having to specifically instrument those systems
  - An Alertmanager, which can process alerts synthesized from out-of-compliance metrics
  - A push gateway to support gathering metrics from short-lived processes

Notes

Prometheus pulls metrics from targets.. The purpose for the push gateway is that short lived processes won't be around for Prometheus to poll, so instead those processes are instrumented to push (send) their data to the gateway, which acts as a cache from which Prometheus can pull the metrics, even after the instrumented process has terminated.

You can instrument applications using the client libraries in order to expose metrics. There are also exporters which can expose metrics for existing applications without manually instrumenting them For example, there is an exporter for Java that will export all JMX metrics to Prometheus.

1.17 Prometheus Architecture



1.18 Service Discovery

- In order to poll targets, Prometheus first needs to find them. This is referred to as Service Discovery (SD).
- Prometheus provides a pluggable approach to Service Discovery. Amongst the options are:
  - File Discovery (file_sd)
  - Kubernetes
  - Consul
  - Azure
  - EC2
  - DNS, and more.
- Prometheus recommends that if none of the existing solutions work for you, that you use the file-based discovery service, and use a tool to write the necessary JSON content.

Notes

Prometheus' authors have gone so far as to declare a moratorium on adding any new Service Discovery implementations, and are promoting the file-based service discovery.

"There is currently a moratorium on new service discovery mechanisms being added to Prometheus due to a lack of developer capacity. In the meantime file_sd remains available." -- https://github.com/prometheus/prometheus/tree/master/discovery

"If you need to use a service discovery system that is not currently supported, your use case may be best served by Prometheus' file-based service discovery mechanism, which enables you to list scrape targets in a JSON file (along with metadata about those targets)." -- https://prometheus.io/docs/guides/file-sd/

https://prometheus.io/docs/prometheus/latest/configuration/configuration/

1.19 File-based Service Discovery

- On the Prometheus server, configure prometheus.yml to define the scrape configuration
- In the scrape configuration, define the file sd configuration. That configuration will include references to JSON files
- The JSON files will define the targets to be scraped.

Notes

A sample prometheus YAML file describing that we are going to poll metrics for a job (type) called "node":

```
scrape_configs:
- job_name: 'node'
  file_sd_configs:
  - files:
    - 'targets.json'
```

That configuration tells us to poll that job using file-based service discovery using a file called "targets.json"

A sample targets.json:

```
[
  {
    "labels": {
      "job": "node"
    },
    "targets": [
      "localhost:9100"
    ]
  }
]
```

That file is what Prometheus recommends you populate using a tool. For example, in a Spring Cloud environment, one could write a tool to populate a JSON file based on information retrieved from Eureka.

1.20 Istio and Prometheus

- Istio ships with support for Prometheus
  - "Mixer comes with a built-in Prometheus adapter that exposes an endpoint serving generated metric values."
  - "The Prometheus add-on is a Prometheus server that comes preconfigured to scrape Mixer endpoints to collect the exposed metrics."
- Istio exposes the following metrics targets: Mixer-generated metrics, Mixer's own metrics, Envoy-generated metrics, Pilot-generated metrics, Gallery-generated metrics, and Policy-related metrics
- If necessary, other metrics targets in an Istio/Kubernetes environment could be discovered using the Kubernetes SD plugin that ships with

Prometheus. However, in many cases this is unnecessary, because …

- Istio is used to implement the Service Mesh Pattern. Therefore, Istio is in a position to generate many metrics for you. You don't need to instrument your services in order to gather metrics related to their service traffic; all of that can be done for you by Istio.

Notes

You can read details on Metrics in Istio at https://istio.io/docs/tasks/telemetry/metrics/.

The section Querying Metrics from Prometheus (https://istio.io/docs/tasks/telemetry/metrics/querying-metrics) discusses the pre-configured nature of the Prometheus add-on for Istio.

The section Collecting Metrics (https://istio.io/docs/tasks/telemetry/metrics/collecting-metrics/) discusses and demonstrates how to get Istio to gather telemetry metrics for your services. The section Collecting Metrics for TCP services (https://istio.io/docs/tasks/telemetry/metrics/tcp-metrics/) discusses and demonstrates how to automatically gather telemetry for TCP services. These two differ only in the set of available attributes available to use when defining the metrics gathering.

Should you have the need, Configuring Prometheus to use Kubernetes SD (https://prometheus.io/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config) covers configuring Prometheus to use Kubernetes SD.

1.21 Exposing Metrics in Services

- Prometheus metrics are defined by their type, value, and name
- Prometheus names metrics in a specific manner, using a name and a set of labels:
- <metric name>{<name>=<value>, ...}

- An example would be:

```
requests_total{service="flights", server="pod54", …}
```

Notes

The structure of Prometheus metric names lets you define a single, common, name for a metric, such as heap_used, heap_free, cpu_usage, requests_per_second, and use the labels (metadata) to filter events later. Each label defines a dimension on which you can query.

1.22 Exposing Metrics in Services

- Prometheus understands four (4) types of metrics, which is stored in its time-series database.
  - Counter: monotonically increasing number
  - Gauge: a number that can increase or decrease
  - Histogram: a set of buckets tracking the number of observations that fall into each bucket
  - Summary: similar to a histogram, but supports quantiles

Notes

Prometheus Metric Types (https://prometheus.io/docs/concepts/metric_types)

A counter is a monotonically incrementing value. It starts at 0, and can only be reset on a restart. Examples of counters would include the number of requests, a running total of bytes, an error count, cars rented, or similar.

A gauge is also a numeric value, but it can increase and decrease. The current occupancy of a resource pool, RAM or CPU use, etc., would be examples of a gauge.

Think of a histogram as you might a bar chart. Each bar (a bucket in histogram terms) represents a classification of a sampled observation, and the value represents the number of such observations. So, for example, you could have a histogram showing the distribution of response sizes. The histogram also tracks the total sum of observation values and the total number of observations.

A summary is very similar to a histogram, but also maintains quantiles over a sliding time window. Quantile are cut points dividing a range of values into sub-ranges of equal size. If quintiles are defined on the range of 0 to 1, the 0.5-quantile would be the median, and the 0.95-quantile would be the 95th percentile.

1.23 Exposing Metrics in Services

- A Prometheus metric is written in plain text.
- The format of an individual metric is:
- Metric name {labels} value optional-timestamp
- To build on our earlier example, a metric might be:

```
response_time{method="post", service="flight",
server="pod54", url="/flights} 13 1560020895000
```

Notes

A metric (https://prometheus.io/docs/concepts/data_model) of the form##
http_response_time{method="post", service="flight", server="pod54", handler="/flights}
13 1560020895000

would be understood as the http_response_time metric, specifically related to a POST to the flight service on server pod54 to the resource handler for the /flights URL with a value of 13(ms), with a UNIX timestamp (optional, and in millisecond precision).

Prometheus has a simple set of recommendations for metric names and labels

(https://prometheus.io/docs/practices/naming/)

1.24 Exposing Metrics in Services

- As a reminder, depending upon your technology stack, you may not need to do anything to expose your metrics. There are existing components that can expose existing metrics from your stack to Prometheus.
  - ~200 (and counting) technologies, including hardware, and software such as Oracle DB, MongoDB, JIRA, Kafka, Kubernetes, and Istio, either have an Exporter or are directly integrated with Prometheus.
    - For JVM-based processes, there is a generic JMX Exporter to export all JMX-based metrics.
    - For microservices written in Python using Django, Nodejs, or Spring Boot w/Hystrix, there are 3rd party libraries that can integrate those services directly into Prometheus on your behalf.
- For systems and/or metrics for which support isn't already provided, you would provide your own, using the client library for any of the 20 or so languages currently supported.

Notes

For a list of Exporters provided for Prometheus, see Exporters and Integrations (https://prometheus.io/docs/instrumenting/exporters). That page also lists software that is directly integrated with Prometheus.

Prometheus officially provides Client Libraries (https://prometheus.io/docs/instrumenting/clientlibs) for Java, Python, Go and Ruby, but there are client libraries for over a dozen other languages.

1.25 Exposing Metrics in Services

- You use one of the Prometheus Client Libraries to define and expose metrics in your application.
- A Client Library provides the infrastructure to define metrics, as well as the HTTP infrastructure allowing it to be polled by Prometheus.
- This is a Spring Bean instrumented with a Prometheus metric, using Spring AOP support in the official Java Client:

```
@Controller#public class MyController {#
@RequestMapping("/")#  @PrometheusTimeMethod(#    name =
"my_controller_path_duration_seconds",#    help = "Some
helpful info here")#  public Object handleMain() {…}
```

- See the student notes for a complete example in Python.

Notes

This sample Python Application copied from the Python Client Library (https://github.com/prometheus/client_python) exposes a simple Summary metric:

```
from prometheus_client import start_http_server, Summary
import random
import time

# Create a metric to track time spent and requests made.
REQUEST_TIME = Summary('request_processing_seconds', 'Time spent processing request')

# Decorate function with metric.
@REQUEST_TIME.time()
def process_request(t):
```

```
    """A dummy function that takes some time."""
    time.sleep(t)

if __name__ == '__main__':
    # Start up the server to expose the metrics.
    start_http_server(8000)
    # Generate some requests.
    while True:
        process_request(random.random())
```

1.26 Exposing Metrics in Services

- The official Java Client library provides:
  - Classes to define any of the four types of Prometheus metrics
  - Support for Spring Framework allowing us to instrument methods with a Summary metric using Spring AOP and a decorator.
  - Collectors – these are objects that will collect for you metrics from the JVM, JMX, logging frameworks, Guava cache, Hibernate SessionFactory, and Jetty Server.
    - The client library also provides a framework allowing you to write custom collectors for code that you cannot instrument
- On the prior slide, we gave an example using Spring AOP
- Likewise, there are multiple ways to expose metrics to Prometheus. The client library includes multiple implementations, including:
  - HTTPServer – embedded HTTP server

- MetricsServlet – a servlet to use with a Java Web Container
- PrometheusMvcEndpoint – an extension for the Spring Boot Actuator
- MetricsHandler – a Virt.x handler to expose metrics

Notes

The official Java Client Library (https://github.com/prometheus/client_java) provides many ways to define metrics and expose them.

This article, https://www.callicoder.com/spring-boot-actuator-metrics-monitoring-dashboard-prometheus-grafana, goes into more detail on the Actuator endpoint provided for Prometheus.

1.27 Exposing Metrics in Services

- Normally, Prometheus polls your service to read its metrics. But what if your process is short-lived?
- As mentioned earlier, and shown on the Architecture slide, Prometheus provides a Push gateway that acts as a cache. Short-lived processes can proactively publish their metrics to the gateway, and Prometheus will pull the metrics from the gateway.
- The Java Client Library provides a PushGateway class so that short-lived processes can push their metrics to the Push gateway.

Notes

See https://github.com/prometheus/client_java#exporting-to-a-pushgateway for an example in the official documentation.

1.28 Querying in Prometheus

- Prometheus provides a bespoke language, PromQL, for querying the time-series database. The elements of PromQL include:
  - Data types
    - instant vector – a set of time series values all sharing the same timestamp
    - range vector – a set of time series containing a range of data points over time
    - scalar – a single, floating-point, value
    - string – a string value
  - Time Series Selectors
    - Instant Time Series Selectors
    - Range Vector Time Series Selectors
    - Offset Modifiers
  - Subqueries – runs an instant query for a given range and resolution, resulting in a range vector.
  - Operators – many logical and function operators, such as comparison and math.
  - Functions – built-in functions that can operate on time series data

Notes

Querying Prometheus (https://prometheus.io/docs/prometheus/latest/querying/basics/) is a primer on the Prometheus Query Language PromQL.

1.29 Querying in Prometheus

- Prometheus queries can be:
  - Simple time series selection – just the name of a metric, or a metric with a set of labels and values to match.
  - Subquery – a query containing a subquery, such as querying the rate based on totals over a period of time.
  - Queries that use PromQL functions and/or operators – an example of this might be a query that fetched a max heap size and the average used heap size over the past 30 minutes, and then reported the average amount of free heap in that period.
- Queries can match string values using regular expressions.

Notes

Querying Prometheus (https://prometheus.io/docs/prometheus/latest/querying/basics/) is a primer on the Prometheus Query Language PromQL.

1.30 Querying in Prometheus

- Examples:
  - Retrieve http_requests_total metrics#http_requests_total
  - Retrieve specific http_requests_total metrics over a 5 minute range#http_requests_total{method="GET", handler="/flight"}[5m]
  - Return the 5-minute rate of the http_requests_total metric for the past 30 minutes, with a resolution of 1 minute#rate(http_requests_total[5m])[30m:1m]
  - Compute the per-second rate for all time series with the

http_requests_total metric name, as measured over the last 5 minutes#rate(http_requests_total[5m])

- Compute the per-second rate for all time series with the http_requests_total metric name, as measured over the last 5 minutes, then reduce to a single value per job name:#sum(rate(http_requests_total[5m])) by (job)

Notes

Querying Prometheus (https://prometheus.io/docs/prometheus/latest/querying/basics/) is a primer on the Prometheus Query Language PromQL.

1.31 Grafana

- Grafana can lay claim to being the leading Open Source solution for time series analytics.
- Grafana allows you to bring data from various sources, including Elasticsearch and Prometheus, and visualize them with beautiful graphs.
  - Yes, Prometheus ships with its own GUI, but Grafana is vastly more capable, and can also handle multiple sources of data, of which Prometheus metrics would be just one.
- Grafana also lets you set alert rules based on the monitored data. When an alert rule fires, it can notify you over multiple channels.
- Grafana dashboards can be quickly designed and shared.
  - Some of the most popular dashboards in the Grafana community are for Prometheus.

Notes

This article, https://www.callicoder.com/spring-boot-actuator-metrics-monitoring-dashboard-prometheus-grafana, goes into more detail on the Actuator endpoint provided for Prometheus, and then uses Grafana to visualize the data.

1.32 Grafana

- Grafana can be broken down into a few functional areas
  - Visualization – Grafana uses panel plugins to provide many different ways to visualize metrics
  - Unify Data Sources – Grafana uses a plug-in architecture to query data sources. Out of the box, over 30 different source types are supported.
  - Alerting – Grafana allows you to write rules about metrics. Those rules are then evaluated
  - Notification – Grafana sends notifications from alerts, supporting many transports, such as Slack, PagerDuty, e-mail, and more.
  - Share – 100s of predefined dashboards and plug-ins are available through Grafana's library.

1.33 Grafana

- Grafana has out of the box support for Prometheus.
  - Prometheus is supported as one of the data source types
  - Grafana provides a PromQL query editor with metric name lookup
  - Predefined, templated, dashboards for Prometheus

- The ability to use aliases for time series names, allowing shorter display names

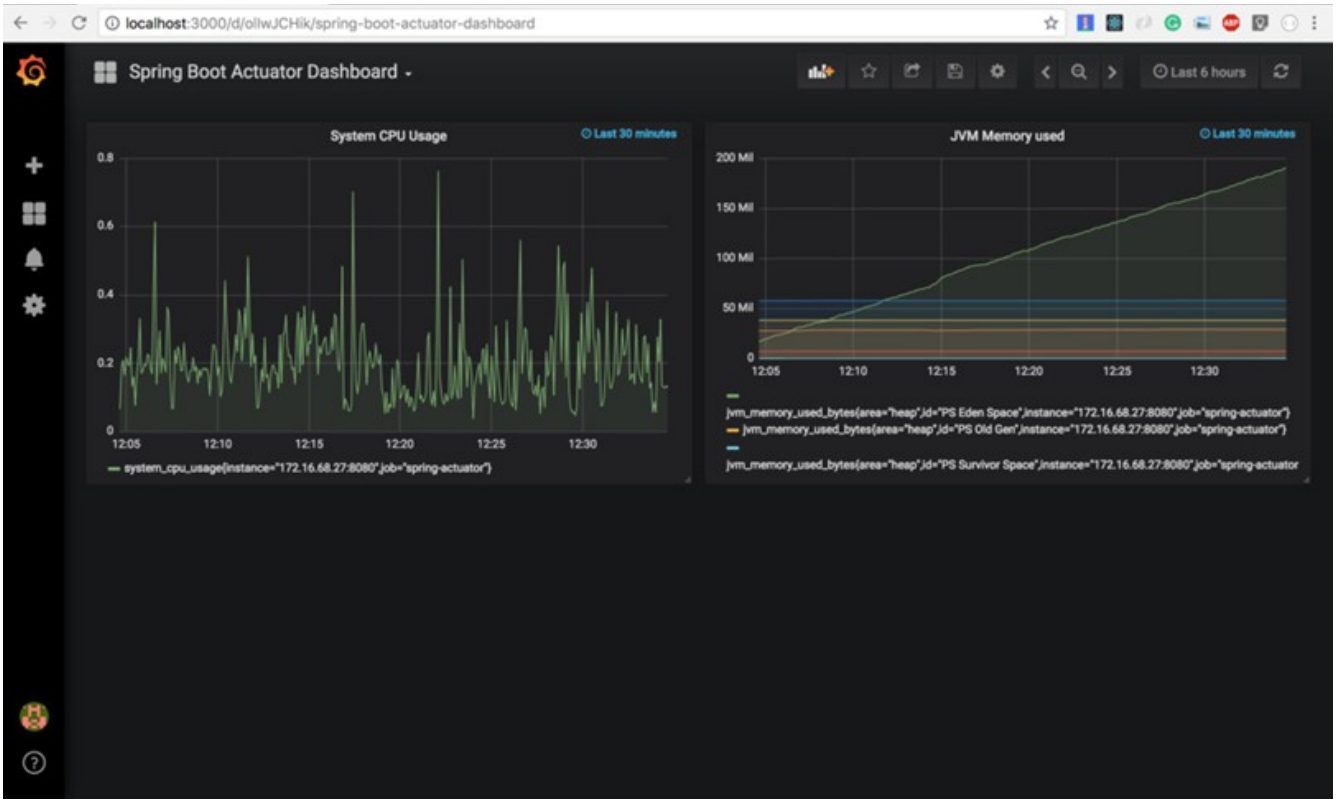
- In addition, not only can Grafana visualize Prometheus metrics, Grafana can expose its own metrics to Prometheus.
- The next few slides show sample Grafana dashboards that are visualizing Prometheus metrics.

1.34 Grafana

- A sample Grafana Dashboard for Prometheus

1.35 Grafana

- A sample Grafana Dashboard for Prometheus



1.36 Grafana

- A sample Grafana Dashboard for Prometheus

This sample comes from Prometheus' documentation for Grafana (https://prometheus.io/docs/visualization/grafana/)

1.37 Business Metrics

- Developers often focus on metrics that relate primarily to process performance
  - Example: ops / sec, resource (CPU, RAM, pools, etc.) use
- Business Analysts care about metrics that correspond to business goals. These are commonly known as Key Performance Indicators (KPI).
  - For example, the average amount of damage to rental vehicles
  - KPIs are often synthesized from time-series data, rather than being recorded as a single metric. Synthesis may happen long after the metrics were recorded, as previously unrecognized relationships are discovered by Business Analysts.
- Coverage of this topic is currently out of scope for this unit, but the student notes include useful supplemental material.

Notes

A short presentation on the topic of visualizing KPIs with Promethus and Grafana:
https://www.slideshare.net/vasster/business-metrics-visualization-with-grafana-and-prometheus

A much longer talk, From Technical Metrics to Business Observability:
https://www.slideshare.net/roidelapluie/prometheus-from-technical-metrics-to-business-observability

1.38 Metrics Using Prometheus Summary

- Overview
- Prometheus
- Service Discovery
- Exposing Metrics in Services
- Querying in Prometheus
- Grafana

1.39 Tracing Using Jaeger

- OpenTracing and its Fundamental Concepts: span and trace
- Jaeger – a Distributed Tracing System, from Über
- Jaeger Client Libraries
- Agent
- Collector
- Query
- Ingester

1.40 OpenTracing

- Tracing is a process of logging the path through code that a request takes. Tracing can help pinpoint failures and performance issues.
- In the case of a distributed architecture, such as a microservices architecture, this takes on both greater import and greater challenge.
  - Our “applications” are really a network of inter-connected microservices
  - When we have many such intertwined microservices working together, it’s difficult to map their inter-dependencies, and to understand the execution of an individual request.
  - Logging does not provide a complete solution.
    - For example, logging alone does not tell us which service was the first in the requests’ flow, nor the sequence taken through the solution topology by the request.
- The solution to this problem is to use a Distributed Tracing technology.

1.41 OpenTracing

- Various vendors had implemented distributed tracing in their own product stacks, but because a microservices solution may be built on many different technology stacks, a unified, vendor and technology neutral, solution is desired.
- The OpenTracing project provides API, frameworks and libraries that allow developers to instrument their code without vendor lock-in.
  - Many of the concepts and vocabulary come from Google’s Dapper project.
  - OpenTracing is vendor-neutral. Jaeger is one implementation of OpenTracing.

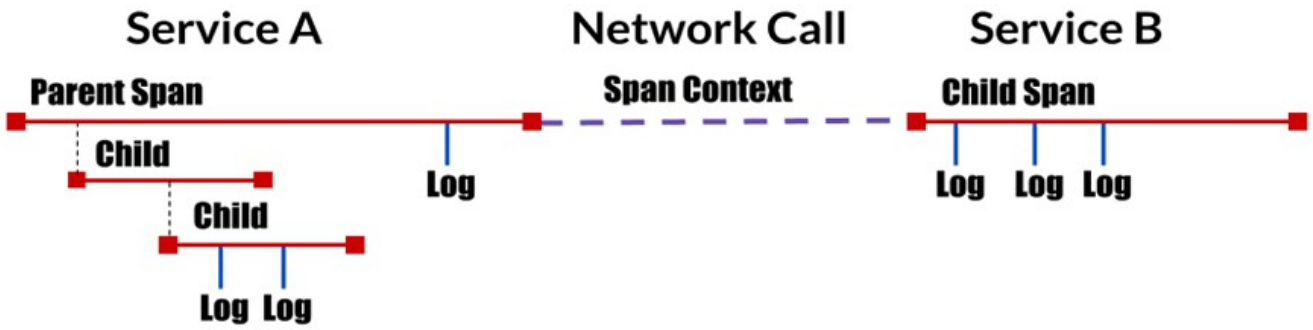
Notes

OpenTraciing Project home page: https://opentracing.io/

Google Dapper white paper: https://ai.google/research/pubs/pub36356

1.42 OpenTracing

- Vocabulary and Data Model
  - Trace – the description of a transaction as it moves throughout the system
  - Span – a named, timed, operation within the overall flow. Each span can have metadata key-value pairs as well as fine-grained, timestamped, logs.
  - Span Context – flows with the transaction containing metadata, such as identifiers so that information can be correlated.



OpenTraciing Specifications: https://github.com/opentracing/specification

1.43 OpenTracing

- The primary concept is the span. A span is a named, timed operation contains the information relating to a single, logical, unit of work.
- As per the specification, a span consists of
  - An operation name
  - A start timestamp and finish timestamp
  - A set of key:value span Tags
    - key:value pairs that enable user-defined annotation of spans in order to query, filter, and comprehend trace data.
  - A set of key:value span Logs
    - key:value pairs that are useful for capturing span-specific logging messages and other debugging or informational output from the application itself
  - A SpanContext, which is propagated throughout the distributed trace
    - implementation-dependent identifiers for the span and trace
    - “Baggage Items” – key-value pairs containing information to be propagated

Notes

OpenTracing Specification (https://opentracing.io/specification/)

1.44 OpenTracing

- A conceptual span example from the OpenTracing Specification

```
span_time=x
end_time=y
operation: db_query

Tags:
- db.instance:"jdbc:mysql://127.0.0.1:3306/customers
- db.statement: "SELECT * FROM mytable WHERE foo='bar';"

Logs:
- message:"Can't connect to mysql server on
'127.0.0.1'(10061)"

SpanContext:
- trace_id:"abc123"
- span_id:"xyz789"
- Baggage Items:
  - special_id:"vsid1738"
```

- The student notes reference an article with multiple span examples for Jaeger

Notes

The Life of a Span (https://medium.com/jaegertracing/the-life-of-a-span-ee508410200b) discusses spans in detail, with multiple examples, and sample code, using Jaeger.

One thing to notice is that Jaeger stores the starting time stamp and the operation's duration.

1.45 Jaeger

- Jaeger, originally developed by Über, is a Distributed Tracing System inspired Apache Zipkin, which was, in turn inspired by Google Dapper.
  - In fact, Jaeger provides support for Zipkin instrumented code, so that applications already working with Zipkin can simply switch to a Jaeger back-end.

Notes

Apache Zipkin: https://zipkin.apache.org/

Jaeger Compatibility with Zipkin: https://www.jaegertracing.io/docs/1.12/features/#backwards-compatibility-with-zipkin

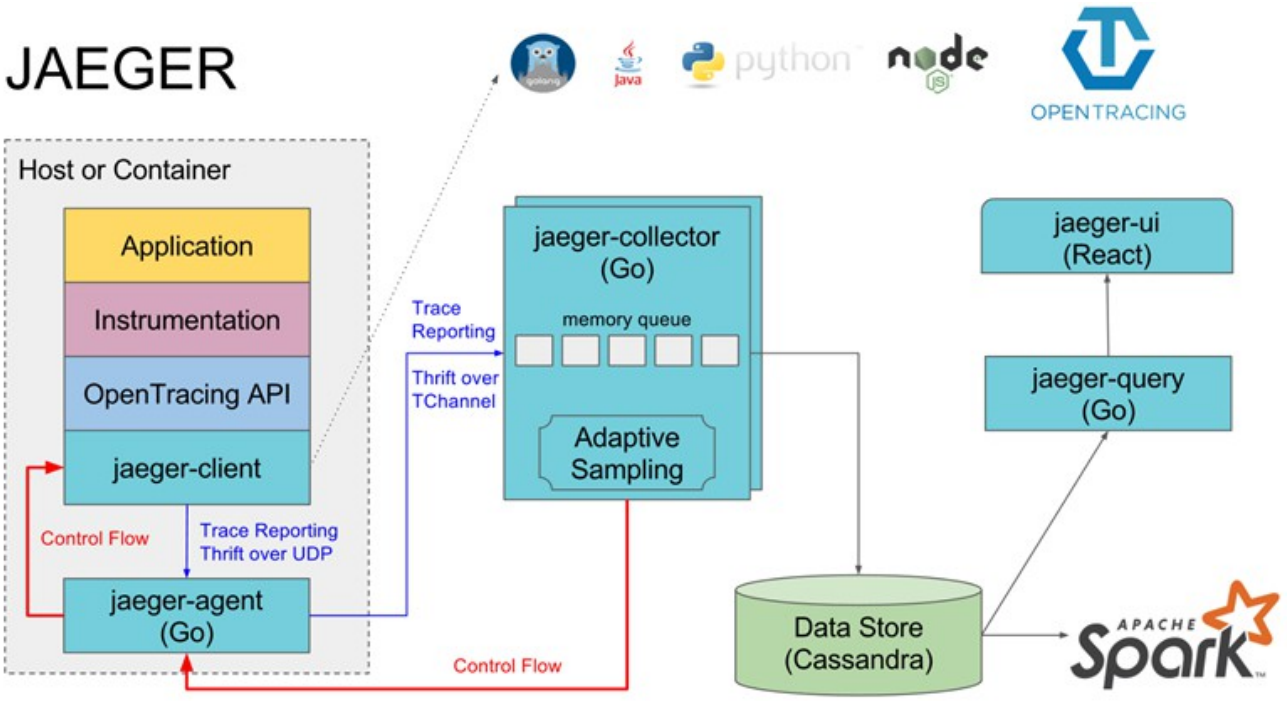
Über published an article explaining how and why Jaeger came about and evolved: https://eng.uber.com/distributed-tracing/

1.46 Jaeger

- The major components that make up Jaeger are:
  - Jaeger Client Libraries
  - Agent
  - Collector
  - Query
  - Ingester

1.47 Jaeger Architecture Diagram

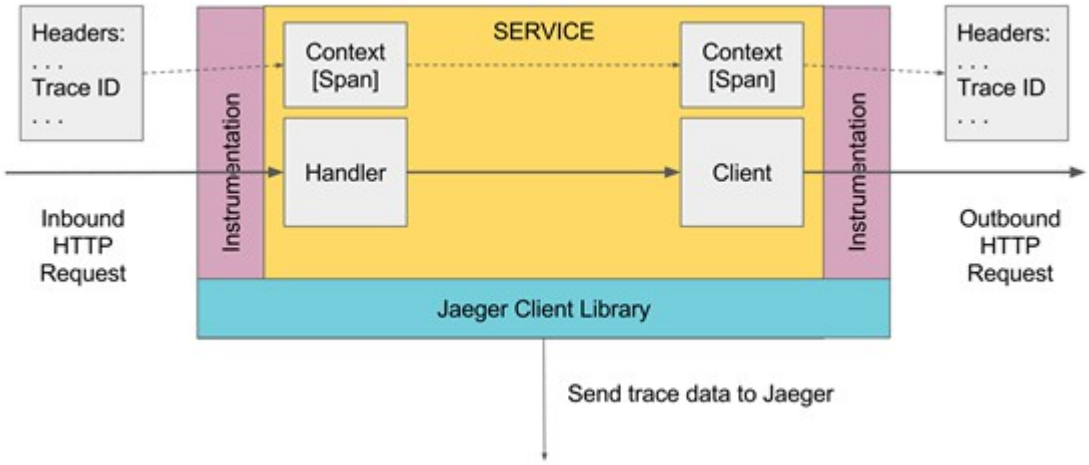
- Jaeger Architecture Diagram



Jaeger Architecture (https://www.jaegertracing.io/docs/1.12/architecture/)

1.48 Jaeger Client Libraries

- Jaeger Client Libraries are language specific implementations of the OpenTracing API.
- Instrumented code use a client library to generate spans, and propagate the SpanContext when invoking other services.



1.49 Jaeger Sampling

- To prevent performance degradation, Jaeger uses the concept of sampling.
- Sampled traces are marked for processing and storage.
  - Once a trace is marked for sampling, or not, that decision is propagated
  - By default, Jaeger samples only 1 trace in 1000.
  - The sampling strategy is pluggable and configurable
    - Istio uses a different strategy. See Student Notes for details
- Spans for sampled traces are emitted from the process.

Notes

Sampling: https://www.jaegertracing.io/docs/1.12/sampling/

Istio Sampling: https://istio.io/docs/tasks/telemetry/distributed-tracing/overview/#trace-sampling

1.50 Jaeger Agent

- The Agent is a network daemon intended to be installed as a sidecar in each container.
- The Agent listens to messages (containing spans) sent using UDP from the Client Library.
- Spans are locally batched, and batches sent to the Collector.

1.51 Jaeger Collector

- The Jaeger Collector service receives messages from Jaeger Agents.
- The messages are placed into a pipeline for processing.
- Messages in the pipeline are validated, indexed, transformed, and stored.
- Collector storage is pluggable. Jaeger currently supports Apache Cassandra, ElasticSearch and Apache Kafka.
  - Kafka is intended to be used as a transport between Jaeger and actual storage.
  - The use of Kafka also opens up additional post-processing opportunities.
- The Jaeger Collector optionally provides support for Apache Zipkin.
  - A Zipkin v1 compatible REST API /api/v1/spans accepts Thrift and JSON

- A Zipkin v2 compatible REST API /api/v2/spans accepts only for JSON

Notes

Collector Zipkin Compatibility: https://www.jaegertracing.io/docs/1.12/getting-started/#migrating-from-zipkin

1.52 Query and Ingester Services

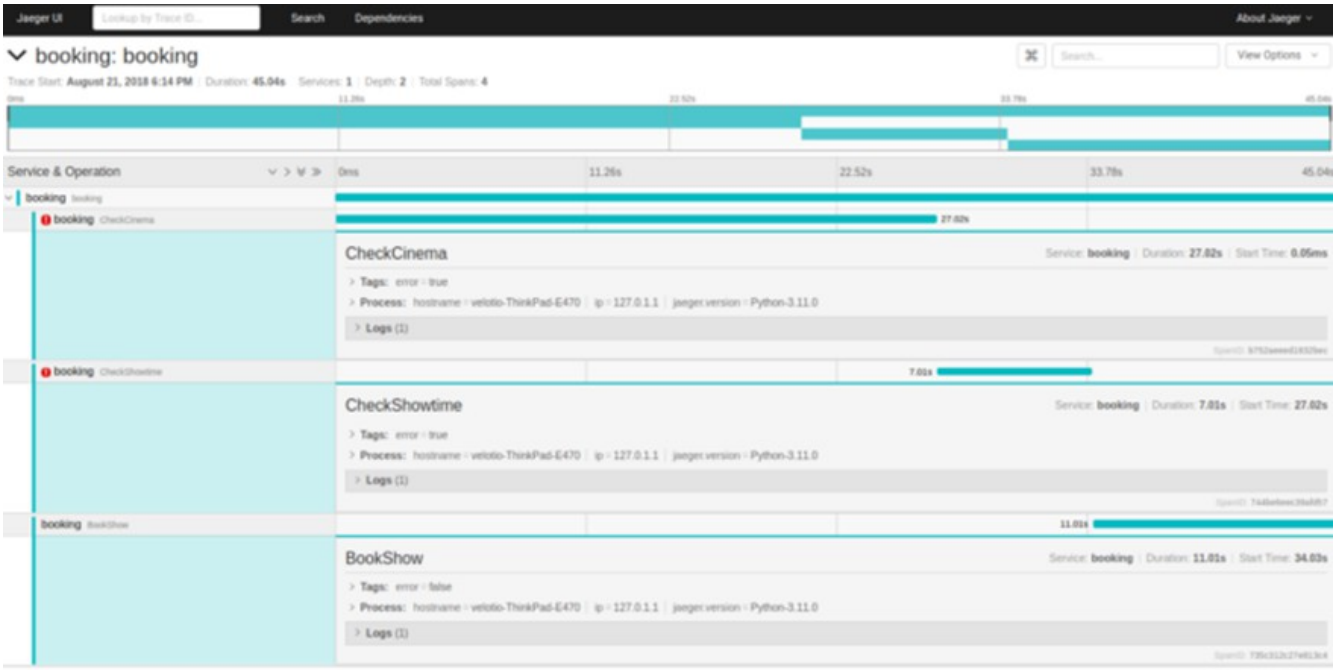
- Jaeger Query is a service that hosts a REST-based query API and provides Jaeger's React-based UI.
- The Jaeger Ingester is a service that reads spans from Kafka topics, and writes them to another backend, e.g., Apache Cassandra or ElasticSearch.

Notes

Jaeger Injester: https://www.jaegertracing.io/docs/1.8/deployment/#ingester

Jaeger Query: https://www.jaegertracing.io/docs/1.8/deployment/#query-service-ui

1.53 Jaeger UI Example



A Comprehensive Tutorial to Implementing OpenTracing With Jaeger (https://medium.com/velotio-perspectives/a-comprehensive-tutorial-to-implementing-opentracing-with-jaeger-a01752e1a8ce) is a useful supplemental resource.

1.54 Jaeger and Prometheus

- Jaeger, itself, being a microservice-based solution, and being part of the Cloud Native Computing Foundation ecosystem, exposes its own metrics using Prometheus.

Notes

Jaeger Metrics: https://www.jaegertracing.io/docs/1.8/monitoring/

1.55 Jaeger and Istio

- Istio supports OpenTracing out of the box, using either Jaeger or Apache Zipkin.
- Istio can automatically generate and send spans on behalf of applications, but applications are required to propagate specific HTTP headers from incoming requests to outgoing requests.
  - Details are in the student notes

Notes

Distributed Tracing (https://istio.io/docs/tasks/telemetry/distributed-tracing/) for Istio covers how to configure Istio for distributed tracing.

Configuring Istio to use Jaeger: https://istio.io/docs/tasks/telemetry/distributed-tracing/jaeger/

The HTTP headers necessary to propagate, and sample source code illustrating the task: https://istio.io/docs/tasks/telemetry/distributed-tracing/overview/#understanding-what-happened

1.56 Tracing Using Jaeger Summary

- OpenTracing and its Fundamental Concepts: span and trace
- Jaeger – a Distributed Tracing System, from Über
- Jaeger Client Libraries
- Agent
- Collector
- Query
- Ingester