jamk.fi

Distributed Microservice Tracing Systems

Open-source tracing implementation for distributed Microservices build in Spring framework

Daniel Mengistu

Master's thesis
February 2020
Technology
Full Stack Software Development

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s) Mengistu, Daniel Mekonnen	Type of publication Master's thesis	Date 04/2020
		Language of publication: English
	Number of pages 74	Permission for web publication: x

Title of publication
Distributed Microservice Tracing Systems
Open-source tracing implementation for distributed Microservices build in Spring framework

Degree programme
Master's degree in Full Stack Software Development

Supervisor(s)
Salmikangas Esa, Huotari Jouni

Assigned by
Company x

Abstract

Monolithic service architectures for enterprise and large backend applications are becoming rare. The monoliths are being replaced with distributed microservice architectures, where the backend application is distributed in an ecosystem of small and specific services.
Company x that assigned this research started to convert legacy codes to microservice. The conversion has brought challenges in finding the source of latency as the number of microservices increase, and the objective of this study is to find these sources of latency.
In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple locations. Each of those services in microservice are dependent on each other directly or indirectly.
This research investigated open sources vender neutral distributed request tracing libraries, namely Brave and Spring Cloud Sleuth. Furthermore, Zipkin and Jaeger, distributed request trace data analyzing, and visualization open source systems were also investigated.
An exemplary microservice architecture was built resembling the microservice architecture in Company x. It is instrumented with tracing libraries, and tracing data are collected and stored in the local storage. Later, data on the collected traces was analyzed and queried for visualization.
It was discovered that the distributed tracing systems are efficient in finding out the source of failure or latency by introducing failure and delay in the service. Zipkin, compared to Jaeger, has been around for quite a long time and has instrumentation library for many languages. However, Jaeger has gained popularity for its easy integration with the latest technologies. e.g. Kubernetes and has large community. Its compatibility with Zipkin instrumentation libraries makes it a preferable distributed tracing library.

Keywords/tags (subjects)
Request tracing, Jaeger, Zipkin, trace, span, openTracing, open-source tracing, tracing library, trace analyse and visualization, microservice, Brave , Spring Cloud Sleuth

Miscellaneous (Confidential information)

Contents

1 Introduction 5
1.1 Background 5
1.2 Motivation And Problem Statement 6
1.3 Research Methodology 7
1.4 Microservice Implementation 7
1.5 Thesis Structure 7
2 Software Development Architectures 9
2.1 Monolithic Architecture 9
2.2 Microservice Architecture 10
3 Distributed Tracing System 14
3.1 Distributed Microservice Tracing Systems 14
3.2 Terminology 16
3.3 OpenTracing 18
3.4 Zipkin 18
3.4.1 Background 18
3.4.2 Zipkin Architecture 18
3.4.3 Instrumentation Library 22
3.4.4 Zipkin Setup 23
3.5 Jaeger 25
3.5.1 Background 25
3.5.2 Jaeger Architecture 25
3.5.3 Jaeger Components 27
3.5.4 Jaeger Setup 29
4 Instrumenting Spring Microservices 33
4.1 Set up Exemplary Microservices Architecture 33

4.2 Adding Instrumentation Library 40

4.3 Distributed Tracing with Zipkin and Jaeger 41

4.4 Visualizing Errors 48

4.5 Detecting Latency 49

5 Discussion 55

6 Further research and development 57

References 58

Appendices 60

Figures

Figure 1. Common example of monolithic architecture 9
Figure 2. Example of microservice with three independent services 11
Figure 3. Anatomy of tracing system 15
Figure 4. Tracing id will not change throughout the process 16
Figure 5. Visualization of a request progress in the process 17
Figure 6. Zipkin Architecture 19
Figure 7. Zipkin UI Component 24
Figure 8. New Zipkin lens UI Component 25
Figure 9. Jaeger direct to storage architecture 26
Figure 10. Jaeger architecture Kafka as an intermediate buffer 26
Figure 11. Jaeger UI component 32
Figure 12. Exemplary microservice architecture 33
Figure 13. Address service main application 35
Figure 14. Address service application property 35
Figure 15. Service discovery Eureka service 37
Figure 16. Eureka service dependency 37
Figure 17. Eureka service property 37
Figure 18. Registry service running, and service are registering 38
Figure 19. Spring cloud gateway dependency 39
Figure 20. Gateway application 39
Figure 21. Gateway application property 39
Figure 22. Adding Spring cloud sleuth library in maven pom.xml file 40
Figure 23. Instrumented library adding information about the service running on MDC 41
Figure 24. Running Jaeger from docker 42
Figure 25. POST request to address-service 42
Figure 27. Zipkin-UI representation of a trace from Address-service 43
Figure 28. Jaeger-UI representation of a trace from Address-service 43
Figure 29. Zipkin-UI span representation from gateway-service 44
Figure 30. Zipkin-UI detail presentation of a trace 45
Figure 31. Jaeger-UI detail presentation of a trace 46

Figure 32. Seven microservices are presented in Zipkin-UI 47
Figure 33. Seven microservices are presented in Jaeger-UI 47
Figure 34. Dependency graph 48
Figure 35. Zipkin-UI representation of a service failure in portal service 49
Figure 36. Jaeger-UI representation of a service failure in portal service 49
Figure 37. Inserting delay in seconds 50
Figure 38. Jaeger-UI shows execution time per service 50

Tables

Table 1. Listing of Zipkin’s v2 API endpoints 21
Table 2. List of instrumentation libraries. 23
Table 3. Jaeger components and ports used 31
Table 4. Address service endpoints.. 34
Table 5. Official language and Library support 53
Table 6. Number of Zipkin's and Jaeger's community and rating 54

1 Introduction

This chapter discusses the objective of the thesis and describes the problems to be solved. It also explains the used technologies and the overall process.

1.1 Background

In the history of application development, applications have been developed or started as one all contained codebase known as monolithic architecture, which is everything from the presentation layer to the database layer in one code set. This approach has a number of setbacks as illustrated in the next chapter; nevertheless, there are also situations where this kind of architecture is necessary. Software developers without a clear understanding and a common way have tried to tackle these problems in their own way until they have ended up using a common style called microservice, an architectural design to build small independent services. Some of the issues emerging in monolitic architecture are e.g. the increased size of the codebase as more and more added features, which also increases complexity and making changes becomes difficult.

Company X, which assigned this thesis is among companies that has started to build their application using monolitic architecture, and it has later adapted microservice architecture. Building an enterprise application in microservice style gets positive feedback and many are trying to incorporate it in their applicaion development; however, there are no clear outlines what microservice style is and how to implement it.

Furthermore, microservice architecture has its own unique problems. When a single end-user requests to come to microservice, the request passes from service to service which is possibly written in multiple frameworks and implemented in different languages. This adds challenges for teams to understand what happened in the journey of a request. In addition to inconsistency between services and the nature of distributed systems, during which a request passes multiple hops over the network and has to be handled by multiple independent services, tracking down and fixing bugs inside microservice is tremendously difficult and involves multiple teams.

This is where the the concept of tracking a request in a distributed system comes to play a major role for developers, DevOps and other stakeholders to better understand their service and detect anomalies at an early stage.

1.2 Motivation And Problem Statement

In recent years, microservices have been successfully adopted by big and small companies. An independent research study was conducted by Dimensional Research (2018) about the adaptation of microservices. The key findings in their 2018 Global microservice trending report are listed as follows:

- Record Growth of Microservices. 92 percent of respondents said they increased their number of microservices in the last year. 92 percent expect to increase their use of microservices in the forthcoming year. Agility (82 percent) and scalability (78 per cent) were the top motivators for adopting a microservice.

- Microservices Widely Used Today. 91 percent are using or have plans to use microservices. 60 per cent have microservices in pilot or production. 86 percent expect microservices to be the default architecture within five years.

- Microservices Introduce New Operational Challenges. 99 percent report challenges in using microservices. 87 percent of those in production report microservices generate more application data. 56 percent report that each additional microservice increases operational challenges.

- Microservice Performance Management is Critical to Success. 98 percent of users that have trouble identifying the root cause of performance issues in microservices environments report it has a direct business impact; with 76 percent of those reporting it takes longer to resolve issues. 73 percent report it is harder to troubleshoot application performance problems in a microservices environment compared to a traditional monolith. 74 percent plan to increase their microservice performance management investment next year.

As the report (Dimensional Research 2018) shows, performance issues have a direct impact on business. Tracking request and monitoring microservices can be a key factor to detect a service failure sooner. This thesis should be able to answer the following questions:

1. How to trace a request in distributed microservices applications ?

2. How to instrument a distributed microservices application?

3. How to find latency in microservices using a distributed tracing system?

4. What are the available open-source distributed microservice tracing systems and criteria for choosing one?

1.3 Research Methodology

In order to achieve the aim of this thesis, distributed microservice based applications are considered. Then, the applications are instrumented by open-source distributed tracing instrumentation libraries and tracing data is collected, aggregated and analyzed using the two widely used open-source distributed tracing systems Zipkin and Jaeger.

1.4 Microservice Implementation

Microservices can be implemented in many languages. For the purpose of this thesis, seven independent microservices are used. These microservices are all built in Java Spring framework. Java programming language has a number of matured frameworks, and Spring framework is adapted extensively for the development of microservices.

1.5 Thesis Structure

The second chapter discusses different microservice development architectures, namely monolithic and microservice architecture. Later in the chapter, the major advantages and disadvantages of each architecture are discussed. The study focuses more in detail on microservice architecture and further discusses its characterstics.

The third chapter discusses distributed tracing systems and shows two widely adapted open source tracing systems, namely Zipkin and Jaeger. The terminology, general architecture and the components of each of these distributed systems are discussed.

The fourth chapter discusses how to instrument the researched microservice using different instrumentation libraries and a distributed tracking platform to analyse the collected information.

The fifth chapter discusses the limitation of a distributed tracing system and compares and contrasts the two open source distributed tracing systems. A conclusion is presented as well.

2 Software Development Architectures

This chapter introduces the two common software development architectural patterns, Monolithic and Microservice architectures. The advantages and disavantages as well as the characterstics of microservices are elaborated as it is the main focus and objective of the thesis. Further, the chapter explains what distributed microservice tracking is.

2.1 Monolithic Architecture

A typical enterprise application often has three main parts: one is the client side part which consists of HTML pages and JS running on the user machine. The second is database, which stores data and backend application containing the business logic and handling all HTML requests, process business logic, retriving and updating database and send responce to the client. The backend which is also called the server-side can be implemented in modules as well. When an application handles all the logic excution and is packed as a single unit, it is called monolithic. As Figure 1 shows, the client side, business logic with two modules for different purposes and the data access layer are all bundled in a single application.

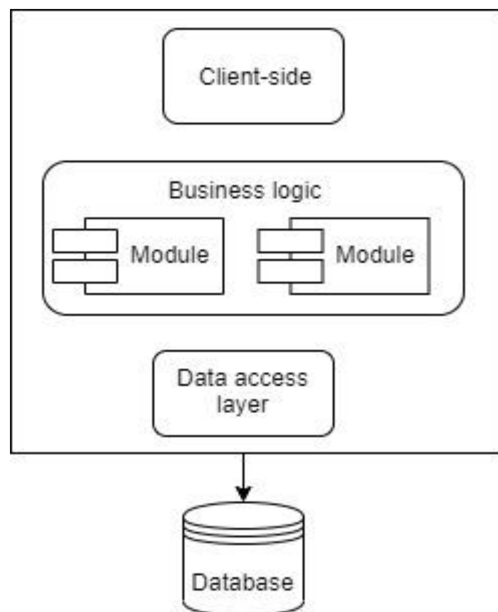

Figure 1. Common example of monolithic architecture

Monolithic architecture has its own advantages and also some drawbacks. When starting to develop an application, monolithic architecture is simple and easy. This kind of application is also easy to test and deploy. Testing can be set up to go through each module from the start to the end. Even if monolitic applications are built in modules, they are deployed as one package depending on the application's language and the used framework. Therefore, deploying them is also simple. This approach at the early stage of application development has no sign of limitation. As the application grows and every time when a new feature is added, only then one starts to experience its limitations.

When a monolithic application is getting large, in a sense that it is performing many tasks and has many features, adding one feature significantly increases the application complexity and decreases reliability significantly as one bug can take down the whole application. The complexity increases time to understand the application and easily find bugs and fix it.

Every small update needs the whole application to be rebuilt, repacked and deployed. Consequently, continuous deployment is problematic. As the size of the application increases, it will also slow down the application start up.The alternative to this architecture is microservice architecture.

2.2 Microservice Architecture

The term "microservice" was discussed at a workshop of software architects near Venice in May, 2011 (Fowler & Lewis 2014) to describe what the participants saw as a common architectural style that many of them had been recently exploring. In March 2012 (James 2012), the same group decided on the term "microservices" as the most appropriate name. James Lewis (2012) presented some of microservices ideas as a case study in March 2012 at 33rd Degree in Krakow in Microservices - Java, the Unix Way as did Fred George (2013) at about the same time. Adrian Cockcroft at Netflix describing this approach as "fine grained SOA" pioneered the style at web scale as were many of the others mentioned in this article - Joe Walnes, Daniel Terhorst-North, Evan Botcher and Graham Tackley (Fowler & Lewis 2014).

The microservice architectural development is an approach to develop an application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by a fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. (Fowler & Lewis 2014)

Each microservice is a small application that has its own hexagonal architecture consisting of business logic along with various adapters. Microservices consume and also expose REST API or other intercommunication mechanisms.

Microservice architecture has a number of advantage when compared to monolithic architecture. By splitting an application to a set of basic services specific to a task, the complexity is minimised, it is easier to understand, development of new features is sped up and the features are deployed independently. With microservice it is possible to choose any suitable programming language and technologies suitable for each microservice. Figure 2 illustrates the idea of a microservice that has three independent services with different database adapters and with a different API format intercommunicating.

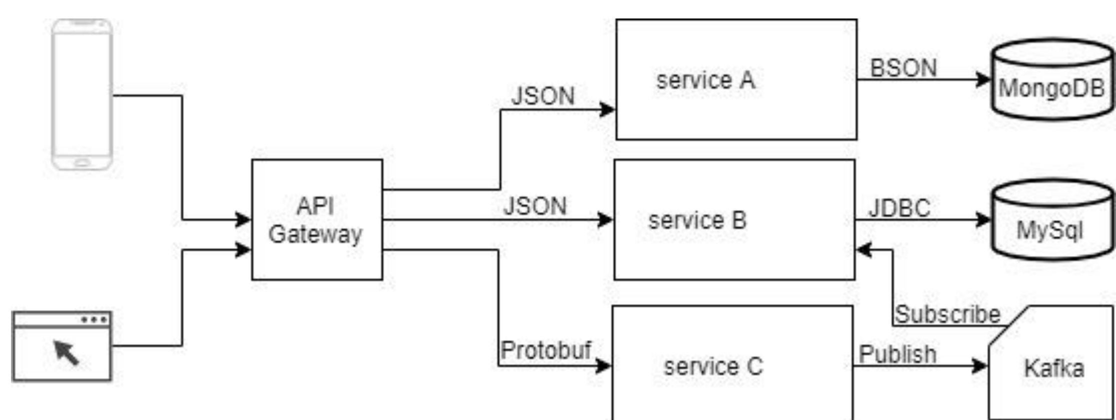

Figure 2. Example of microservice with three independent services

The Microservice architecture pattern significantly affects the relationship between application and database. In microservice architecture, instead of sharing a single

database schema like monolithic architecture with other services, each service has its own database schema. Some argue that this approach is at odds with the idea of an enterprise-wide data model. Additionally, it often results in duplication of some data. However, when having a database schema per service it is essential to benefit from microservices because it ensures loose coupling. Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs. (Kharenko 2015)

Just like monolithic architecture and other technologies, microservice architecture has its own setbacks. Microservices are distributed systems. For distributed systems to function properly, the developer needs to consider inter-process communication and also handle partial failure of requests since the destination of a request might be slow or unavailable. This adds complexity to microservice architecture. Another point is that microservice uses partition database, which is a division of a logical database or its constituent elements into distinct independent parts and a transaction updating multiple entities may need to update multiple databases owned by different services. Testing the whole application takes more work than with a monolithic approach. Even if there is no official definition of microservices, many developers agree on the common characteristics exhibited by microservice architectures. James Lewis and Marin Fowler explained in their blog the following common characteristics of microservice: (Fowler & Lewis 2014)

Componentization via microservices: The componentization of functionality in a complex application is achieved via services, or microservices that are independent processes communicating over a network. The microservices are designed to provide fine-grained interfaces and be small-sized, autonomously developed, and independently deployable.

Smart endpoints and dumb pipes: The communications between services utilize technology-agnostic protocols such as HTTP and REST, as opposed to smart mechanisms like the Enterprise Service Bus (ESB).

Organized around business capabilities: Products, not projects e.g. the services are organized around business functions ("user profile service" or "fulfillment service"), rather than technologies. The development process treats the services as

continuously evolving products rather than projects that are considered completed once delivered.

Decentralized governance: Allows different microservices to be implemented using different technology stacks.

Decentralized data management: Manifests in the decisions for both the conceptual data models and the data storage technologies being made independently between services.

Infrastructure automation: The services are built, released, and deployed with automated processes, utilizing automated testing, continuous integration, and continuous deployment.

Design for failure: The services are always expected to tolerate failures of their dependencies and either retry the requests or gracefully degrade their own functionality.

Evolutionary design: Individual components of a microservices architecture are expected to evolve independently, without forcing upgrades on the components that depend on them.

3 Distributed Tracing System

Chapter 3 explains the concept of a distributed microservice tracing system and what it is composed of, what is distributed tracking system architecture and what are open-source distributed microservice tracing systems available.

3.1 Distributed Microservice Tracing Systems

Microservice architecture adaptations by small and large backend development have grown in recent years. Microservice has several advantages over monolithic architecture; however, it did not come without added complexity. This is specifically the case when tracking a request, analyzing, and pointing out where the bottleneck has occurred as a request passes from one service to another.

Distributed microservice tracing is all about following the path a request has taken starting from front-end or mobile application to every microservice it passes to the database and back. Distributed tracking can be used by DevOps to monitor the application performance. It is also used by developers for debugging and optimizing their code.

Modern application development is inclined to the cloud native and microservice architectural development. Currently there are many commercial and open source distributed systems. Many distributed tracing solutions are either based on, or inspired by, the Google Dapper Whitepaper. (Sigelman, Barroso, Burrows, Stemphenson, Plakal, Beaver, Jaspan & Shanbhag 2010)

Distributed systems are mainly composed of five components. The first one is instrumentation of application and middleware that is to incorporate tracing library to a microservice. Data is collected at instrumenting tracing points in the client application. A request can be instrumented with two tracing points: when a request comes to a service and before passing on to the next service. The collected data in those tracing points is collectively called trace.

The second among five components which make up distributed system, is distributed context propagation. Several solutions have been proposed to correlate traces from different services for a request. Context propagation is a way where tracing points

give a global identifier for the data they produce, which is unique for each traced request, and passing the global execution identifier needs to be passed along the execution flow. Then later by grouping traces by their identifier, the whole request execution path can be reconstructed.

Thirdly comes trace ingestion. It is a way traces are passed to the collection. The Tracing API is implemented with a concrete tracing library that reports the collected data to the tracing backend, usually with some in-memory batching to reduce the communications overhead. Reporting is always created asynchronously in the background, off the critical path of the business requests (Shkuro 2019a). Figure 3 illustrates this microservice tracing anatomy(Adapted from Shkuro 2019a). When a request arrives or leaves a microservice, metadata and other operation related info is added to the trace. The instrumented library collects traces and passes them on to the collector. The collector persists traces to a database, and traces are analyzed then by tracing UI display traces.

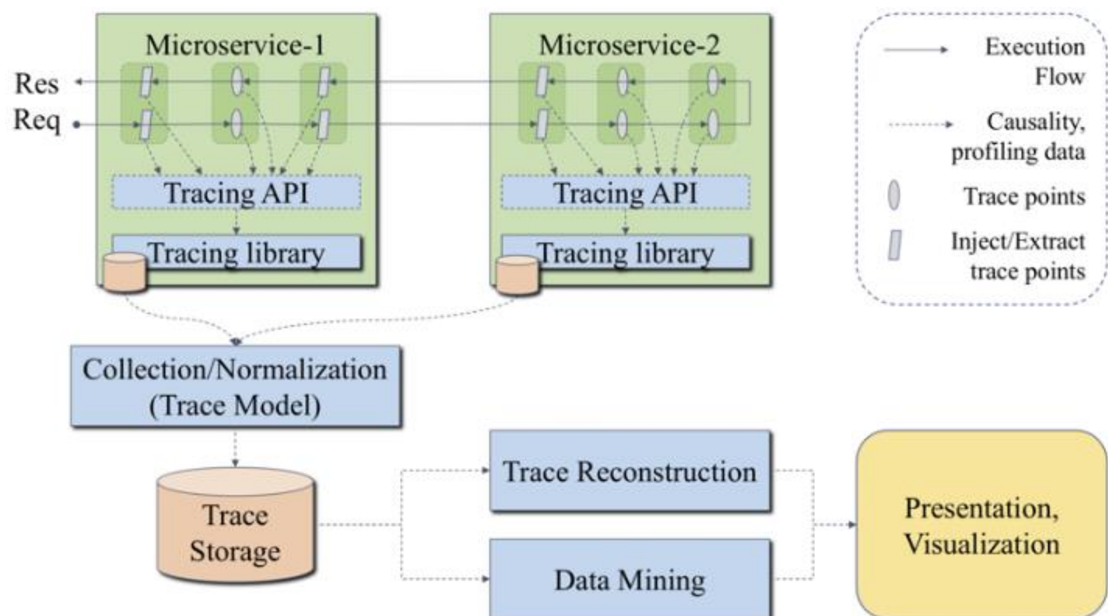

Figure 3. Anatomy of tracing system

3.2 Terminology

Span

Span is the smallest logical unit of a distributed system. It is the primary building block of the distributed system. Every microservice where the request passes, contributes a span describing operation and duration. Different distributed microservice systems have different contents in the span. According to openTracing API (Spans 2020), specification spans contain an operation name, start and finish timestamp, span context which carries data across process boundaries and key-value pair called tags and logs. Tags enable user-defined annotation of spans in order to query, filter, and comprehend trace data. Logs are useful for capturing *span-specific* logging messages and other debugging information. (Spans 2020)

Trace

Trace covers the entire request across all services it touches. It consists of all the spans for a request. Trace is built after a request has been completed. Figure 4 illustrates this: When a request comes to microservice, a unique tracing ID is produced, and it will not change when it passes to the next microservice. Span measures each operation in a microservice. Normally, a microservice has multiple spans. During the journey of the request, it can create one or multiple spans.

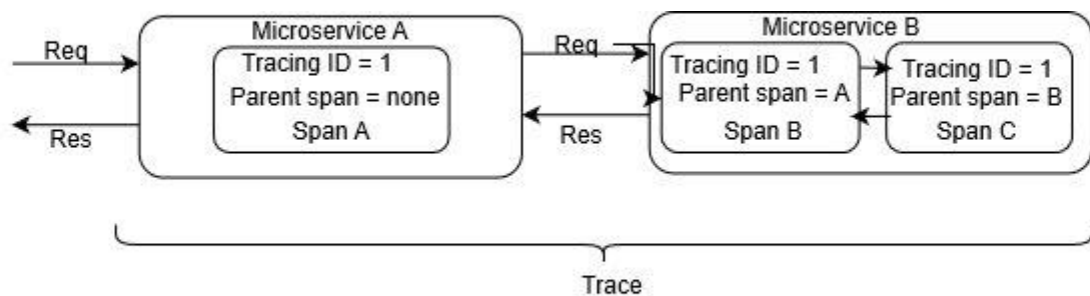

Figure 4. Tracing id will not change throughout the process

Baggage

Baggage is a context propagation mechanism used by OpenTracing API to capture, propagate and retrieve distributed metadata. It is a collection of key/value pairs that are defined and used by the distributed tracing system. Distributed context propagation is a generic mechanism that can be used for purposes completely unrelated to end-to-end tracing. In the OpenTracing API it is called “baggage”, a term coined by Prof. Rodrigo Fonseca from Brown University (Shkuro 2019b) because it allows to attach arbitrary data to a request and have this data automatically propagated by the framework to all downstream network calls made by the current microservice or component. The baggage is carried alongside the business requests and can be read at any point in the request execution.

Figure 5 describes a Span which is the smallest unit of trace. A microservice can have multiple Spans. Spans have a relationship with each other. There can be one parent Span if it has one or more child Spans. This parent-child Span relationship helps to draw trace tree out of all Spans in one request.

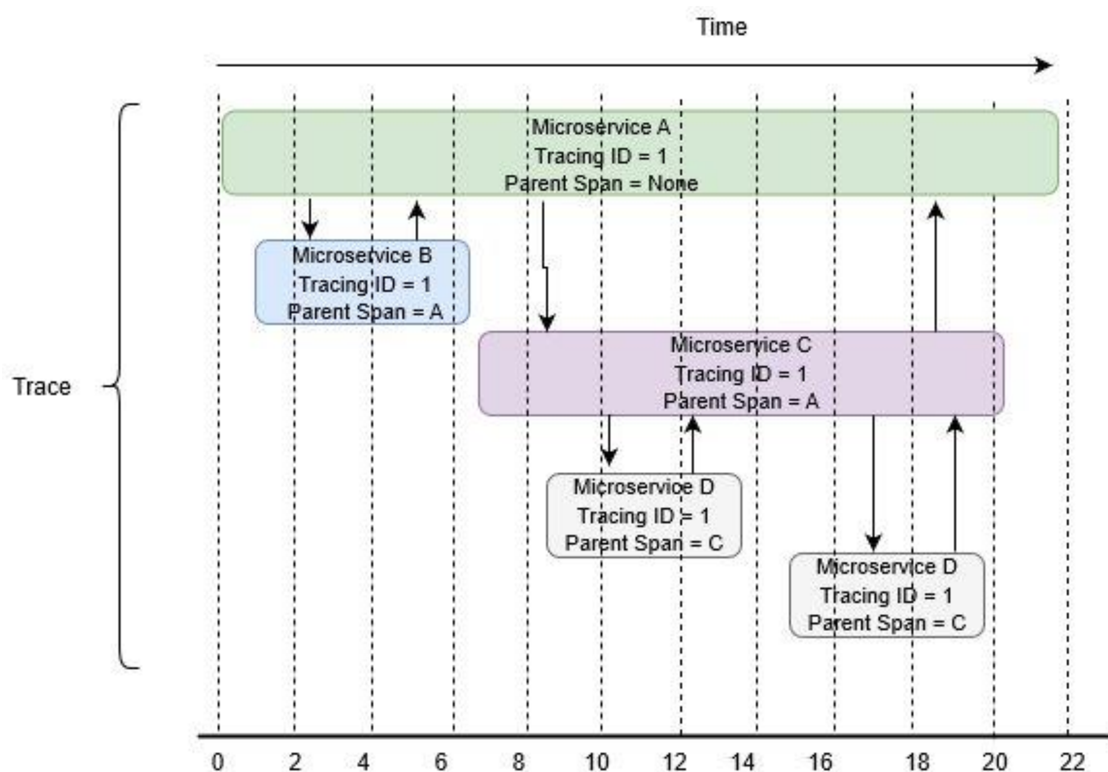

Figure 5. Visualization of a request progress in the process

Tags

Tags are key-value pairs of data associated with recorded measurements to provide contextual information, distinguish and group metrics during analysis and inspection.

3.3 OpenTracing

OpenTracing API tries to provide a standard, a vendor neutral framework for instrumentation. There are different ways to instrument applications with tracing library and there is no standard, so it is difficult to try different distributed tracing APIs. OpenTracing works towards creating more standardized APIs and instrumentation for distributed tracing. OpenTracing is open source and part of the Cloud Native Computing Foundation (CNCF). OpenTracing enables developers to try out a different distributed tracing system without the need to repeat the whole instrumentation process for the new distributed tracing system by simply changing the configuration of the Tracer.

3.4 Zipkin

3.4.1 Background

Zipkin is one of the pioneers and widely used distributed tacking systems. It was originally developed by Twitter using Google paper (Sigelman et a. 2010) The main reason Twitter has developed Zipkin was to gather timing data for all the disparate services involved in managing a request to the Twitter API. Twitter open sourced Zipkin in June 2012 under the APLv2 license. At the time, Twitter described it as “a performance profiler, like Firebug, but tailored for a website backend instead of a browser”. (Distributed Tracing for Microservices)

3.4.2 Zipkin Architecture

An application instrumented with tracing libraries has tracers. Those tracers record operation timing and metadata about the operations. When the operation continues another service, the tracers need to send only a tracer id to relate Spans in the whole

request journey. Instrumented client communicates with a collector in three main transport protocols: HTTP, Kafka and Scribe. Spans are reported to collectors asynchronously so that the communication between collector and reporter does not affect the request or a normal operation.

Zipkin is made of four components: The first one is the collector which collects, validates, index spans reported by instrumented library before sending them to storage (Architecture 2020).

The second Zipkin component is storage which is where the indexed spans are stored. Zipkin by default stores spans in-memory before sending them to the collector, the purpose of which is not to overload the server with too many connection pools. This also means that if the server boots up, all collected data will be lost; hence, some way is needed to store the spans persistently. Zipkin provides in-memory, MySQL, Casandra and Elasticsearch for a scalable storage database option.

Figure 6 (Adapted from Architecture 2020) describes a client microservice instrumented to collect metadata and send spans to Zipkin.

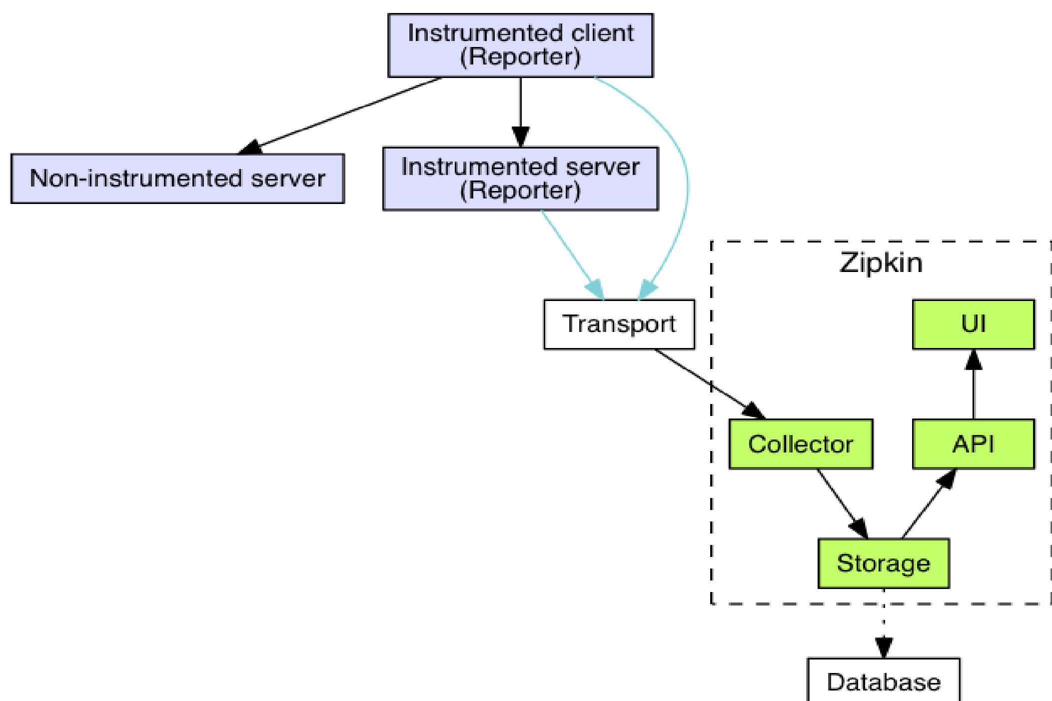

Figure 6. Zipkin Architecture

The third component is Zipkin query API which is an application interface that provides access to the stored spans in the database. Zipkin provides its query end points for developers who want to extend the development as shown in Table 1 which lists Zipkin's v2 API endpoints. (Adapted from Zipkin community 2020 *Zipkin API*)

Table 1. Listing of Zipkin's v2 API endpoints

Get / services	Returns a list of all service names associated with span endpoints.
Get / spans	Gets all the span names recorded by a service
POST / spans	Uploads a list of spans encoded per content-type, for example JSON.
GET / traces	Invoking this request retrieves traces with the endTs, subject limit and lookback parameters. For example, if endTs is 10:20 today, limit is 10, and lookback is 7 days, the traces returned should be those nearest to 10:20 today, not 10:20 a week ago.
GET /trace/{traceId}	Invoking this request with tracing identifier will retrieve all spans with the same tracing identification.
GET /traceMany	Invoking this request retrieves any traces with the specified IDs. This request needs at least two tracing identifiers separated by comma; otherwise it will return a bad request and the results are returned in any order.
GET /dependencies	Returns service links derived from spans. It uses span's metadata about parent span and child span.
GET / autocompleteKeys	Returns a subset of keys from Span.tags conFigured for value autocompletion. This helps sites populate common keys into the annotationQuery parameter of the /traces endpoint. For example, a UI can allow users to select site-specific keys from a drop-down as opposed to typing them in manually. This helps guide users towards the more correct keys and avoids typos or formatting problems.
GET / autocompleteValues	Returns all known values of Span.tags for the given autocomplete key. Refers to the description of /autocompleteKeys for the use case.

The fourth component in Zipkin is UI. Zipkin API provides simple JSON responses for requests coming from Zipkin UI. Zipkin UI is a single page application which provides a method for viewing traces based on service, time, and annotations.

3.4.3 Instrumentation Library

Tracing information is collected by instrumentation library. Instrumentation libraries are different for different frameworks and languages. When a microservice is instrumented by instrumentation library, which is a separate application added to microservice, it collects traces and sends it to Zipkin. When the instrumented microservice makes a request to another microservice, it passes a few information and tracing identifiers along with the requests to Zipkin so it can later tie the data together into spans. Table 2 (Adapted from Zipkin Community 2020, *Tracers and Instrumentation*) shows a list of instrumentation libraries existing for each programming language and framework.

Table 2. List of instrumentation libraries.

Language	Library	Framework	Propagation Supported	Transports Supported
C#	Zipkin4net	Asp.net core, OWIN	HTTP (B3)	Any
Go	Zipkin-go	standard Go middleware	HTTP (B3), gRPC (B3)	HTTP (v2), Kafka (v2), Log
Java	brave	Jersey, RestEASY, JAXRS2, Apache HttpClient, MySQL	Http (B3), gRPC (B3)	HTTP, Kafka, Scribe
JavaScript	Zipkin-js	cujoJS, express, restify	HTTP (B3)	Http, Kafka, Scribe
Ruby	zipkin-ruby	Rack	HTTP (B3)	HTTP, Kafka, Scribe
Scala	zipkin-finagle	Finagle	HTTP (B3), Thrift	HTTP, Kafka, Scribe
PHP	zipkin-php	Any	B3	HTTP, log file

3.4.4 Zipkin Setup

There are three options to run Zipkin in a local machine. Zipkin can run from Docker or if the machine has Java 8 or higher version installed, it can run from self-contained executable jar or by running a source code. Here it is running from Docker as follows:

```
$docker run -d -p 9411:9411 openzipkin/zipkin
```

The -d flag detaches from the terminal and runs the process in the background. The -p flag is to expose ports on the host network that the Zipkin backend is listening to.

When a storage is not configured, Zipkin uses in-memory storage, which is not persistent. By browsing to http://localhost:9411, Zipkin UI will be displayed.

Figure 7 and Figure 8 show the Zipkin-UI component.

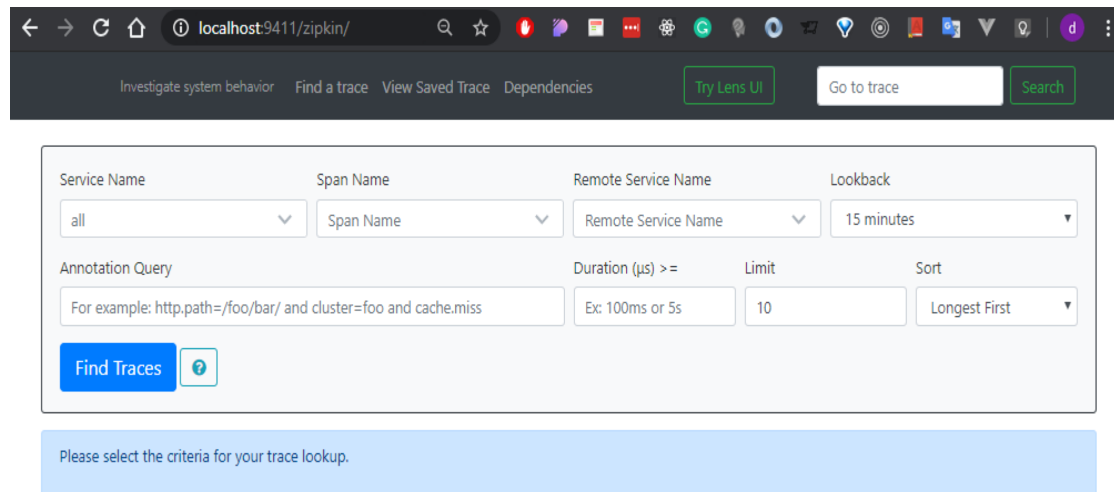

Figure 7. Zipkin UI Component

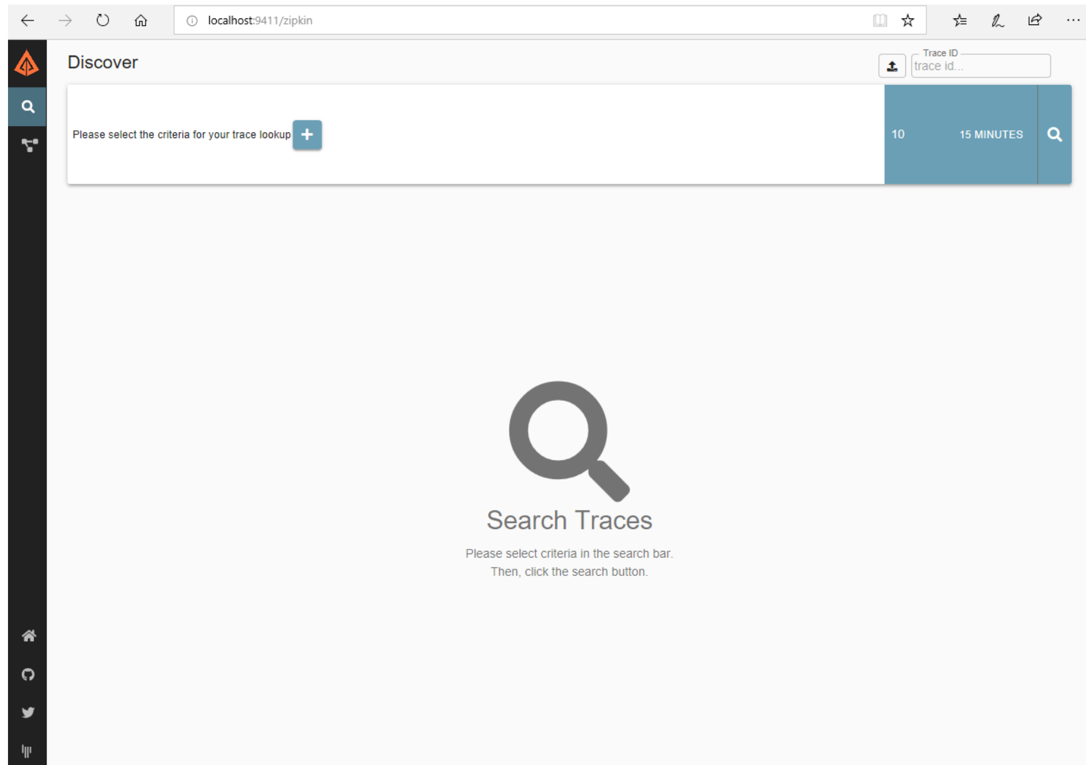

Figure 8. New Zipkin lens UI Component

3.5 Jaeger

3.5.1 Background

Jaeger is an open source distributed tracing system developed by ridesharing company Uber. Jaeger implements vendor-neutral open tracing specifications, and it was accepted as a Cloud Native Computing Foundation (CNCF) Incubation project in 2017 and promoted to graduated status in 2019.

3.5.2 Jaeger Architecture

Jaeger has five components working together to collect, store, query and visualise spans and traces. Jaeger backend components are implemented in Go language. Jaeger has two types of deployment. Figure 9 shows a type of deployment where Jaeger component called Jaeger-collector store traces directly to the storage. (Adapted from Architecture 2020).

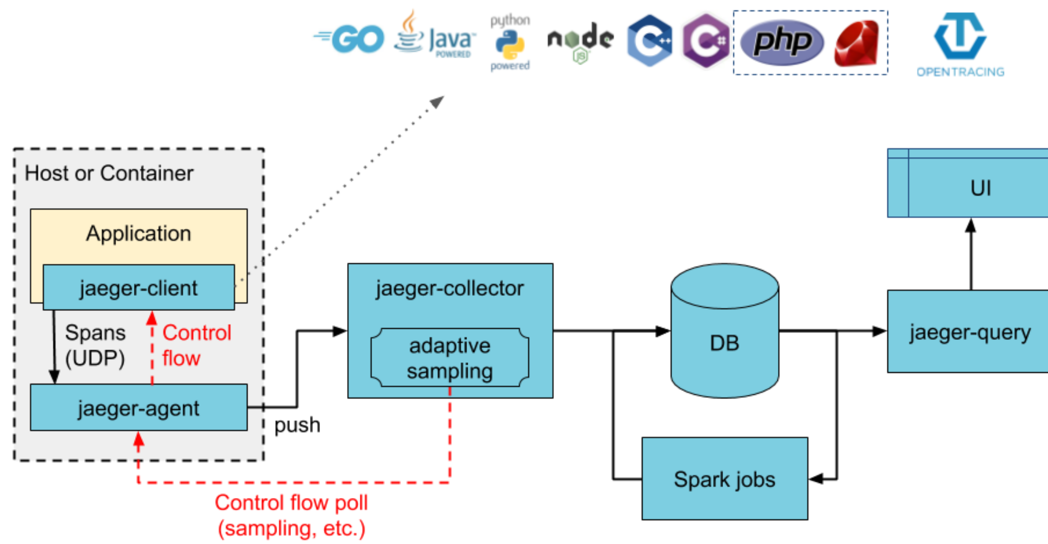

Figure 9. Jaeger direct to storage architecture

The second type of deployment shown Figure 10, where Jaeger collector writes to Kafka. (Adapted from Architecture 2020). Kafka publishes traces and a service which subscribed to it and start to use traces as it published.

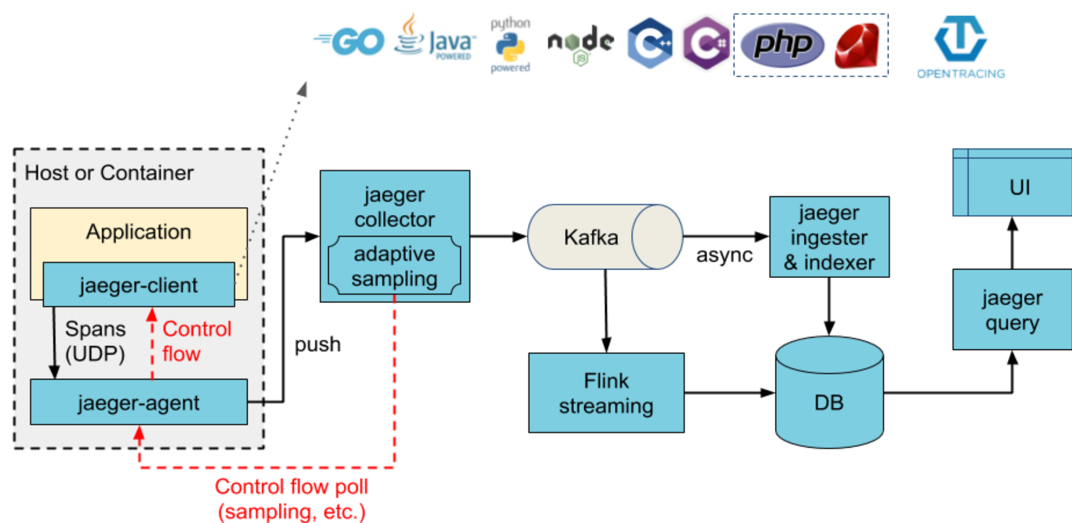

Figure 10. Jaeger architecture Kafka as an intermediate buffer

3.5.3 Jaeger Components

Jaeger Client

Jaeger client is a programming language specific tracing library that resides in a client application to measure and collect information related to the operation. It is an OpenTracing API specification implementation for distributed tracing. When an application gets a request, spans are created, and context information is attached to it. The Jaeger client typically sends spans via UDP to the agent, avoiding the TCP overhead and reducing the CPU and memory pressure upon the instrumented application (Architecture 2020).

When spans propagate, not all information included propagates with it but only trace id, span id and baggage are propagated with the request. Jaeger's Adaptive Sampling is a technique used to record only a subset of all traces. It has several advantages. First, it minimises performance overload on the instrumented application while guaranteeing that minimal traces are collected from the instrumented application with low QPS (queries per second). Secondly, it allows detailed control of sampling strategies per endpoint, rather than on a service-level or global basis.

Jaeger is very similar to Zipkin; however, Jaeger has features not available in Zipkin, including adaptive sampling and advanced visualization tools in the UI. If a service has two endpoints with huge different throughputs, then its sampling rate will be driven on the high QPS endpoint, which may leave the low QPS(Q) endpoint never sampled. For example, if the QPS of the endpoint is different by a factor of 100, and the probability is set to 0.001, then the low QPS traffic will have only 1 in 100,000 chance to be sampled. (Sampling 2020)

Jaeger libraries support four types of sampling. Constant sampler makes the same sampling for all traces. It either samples all trace or none of the traces are sampled. Probabilistic sampling makes a random sampling decision with the probability set in the configuration. The rate limiting sampler uses a certain rate limiter to sample traces. For instance, if it is set to 4, then it will sample 4 requests in a second. Remote sampler is a default sampling strategy that allows controlling the sampling strategies in the services from a central configuration in Jaeger backend, or even dynamically (Architecture 2020).

Jaeger Agent

The Jaeger Agent is a network daemon responsible for receiving spans from a Jaeger client. It communicates through User Datagram Protocol (UDP) and forwards the spans to the next component called Jaeger Collector. The Jaeger agent is an abstraction layer that acts like a buffer to abstract batch processing and routing from the client. One advantage of agents is that they are local to the host, which means clients do not need to carry out any service discovery to find them. Jaeger client's user HTTP endpoints is implemented in the agent to get sampling strategies and other configurations. Jaeger agent runs as Daemon by default; however, in a Kubernetes setup the agent can be configured to run as a sidecar container in the application Pod or as an independent Daemon Set (Architecture 2020).

Jaeger Collector

The Jaeger Collector is responsible for receiving batches of spans from Jaeger Agent, running them through the processing pipeline and storing them in specified storage backend. Through configuration it is also possible that the client can directly communicate with the collector. Before the collected traces are stored, data-processing pipeline validates traces, indexes them and performs any transformations (Architecture 2020).

Jaeger Storage

This component is responsible for storing traces, and it requires a persistent storage backend. Jaeger supports Cassandra and Elasticsearch, and open source NoSQL databases as trace storage backends. It also supports in-memory storage; however, it is meant for short test and quick start up as everything is wiped out at start up (Architecture 2020).

Jaeger Query

Jaeger query service is the back service for Jaeger UI component. It is responsible for retrieving traces from storage and formats them to display on the UI (Architecture 2020).

Jaeger UI

The jaeger UI is a component responsible for visualisation. It is implemented in JavaScript React framework. Jaeger's team developed a new data model that supports key-value logging and span reference. It optimizes the volume of data sent out of process by avoiding process tag duplication in every span. These in turn allow the UI to efficiently deal with large volumes of data and to display traces (Architecture 2020).

3.5.4 Jaeger Setup

To run Jaeger from the Docker image as a prerequisite a running Docker installation is needed in the machine. The following command will run Jaeger's Docker image from the local machine if it exists; otherwise, it will fetch it from the Docker hub.

```
$ docker run --rm jaegertracing/all-in-one:1.8 version
```

The all-in-one Jaeger distribution contains all other components of a normal Jaeger installation, including backend and the web user interface. The other option is to deploy each component separately and configure them to work together. For the thesis experiment purpose, Docker all-in-one image is used, and no database is set up; instead, in-memory is used (Architecture 2020). Using Docker, Jaeger can be run all-in-one with the following command

```
$ docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.8
```

This command allows to set ports for a specific purpose. Jaeger can also accept Zipkin thrift when it is set to listen on port 9411.The --name flag sets a name. The -p flag used to set different ports for a different purpose to expose ports on the host network on which the Jaeger backend is listening. Table 3 describes which ports are used by which Jaeger component and protocol used.

Table 3. Jaeger components and ports used

Service	Port	Protocol	Description
Jaeger-query	16686	HTTP	Exposes API endpoints at /api/* and Jaeger UI at /
Collector	9411	HTTP	Accept Zipkin span in JSON or Thrift
Collector	14268	HTTP	Accept Spans directly from clients
Jaeger-agent	5778	HTTP	Serve configs, sampling strategies
Jaeger-agent	6832	UDP	accept jaeger.thrift over binary thrift protocol
Jaeger-agent	6831	UDP	accept jaeger.thrift over compact thrift protocol
Jaeger-agent	5775	UDP	accept zipkin.thrift over compact thrift protocol

Once the container starts, the UI is accessible from http://127.0.0.1:16686/ in the browser. At first there is no tracing data but as Jaeger is self-tracing if the page is reloaded twice, some tracing data becomes visible. The following Figure 11 illustrates that the Jaeger UI component has found Jaeger query service as it refreshed twice, and it displays tracing data.

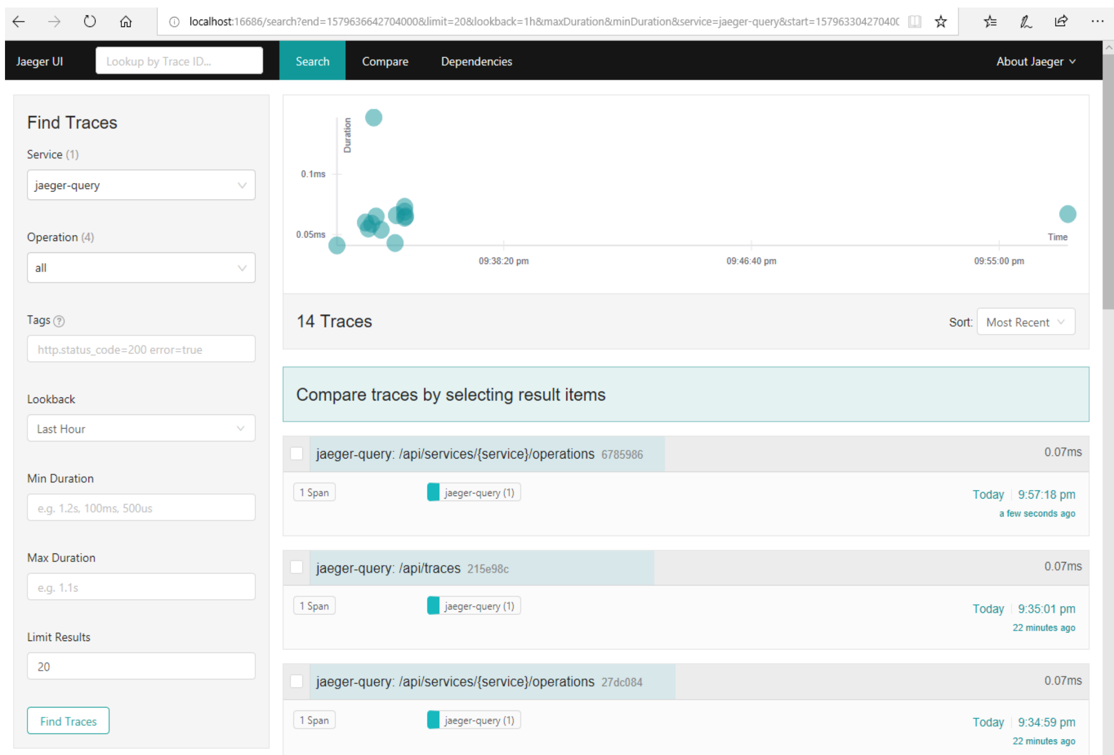

Figure 11. Jaeger UI component

4 Instrumenting Spring Microservices

In this chapter microservice architecture is set up and microservices will be built using Spring framework and Spring Cloud. Later discuss how to add instrumentation libraries which are used as a distributed tracing framework for a microservice architecture in the Spring ecosystem.

4.1 Set up Exemplary Microservices Architecture

The Thesis use eight microservices. Two of them are configurational services and the rest are business related and are built in the same manner. For the sake of simplicity, one of the six microservice, address-service, will be discussed in detail with the other configurational service. Detail code presented in Appendix pages. Figure 12 shows the architecture of the exemplary microservices.

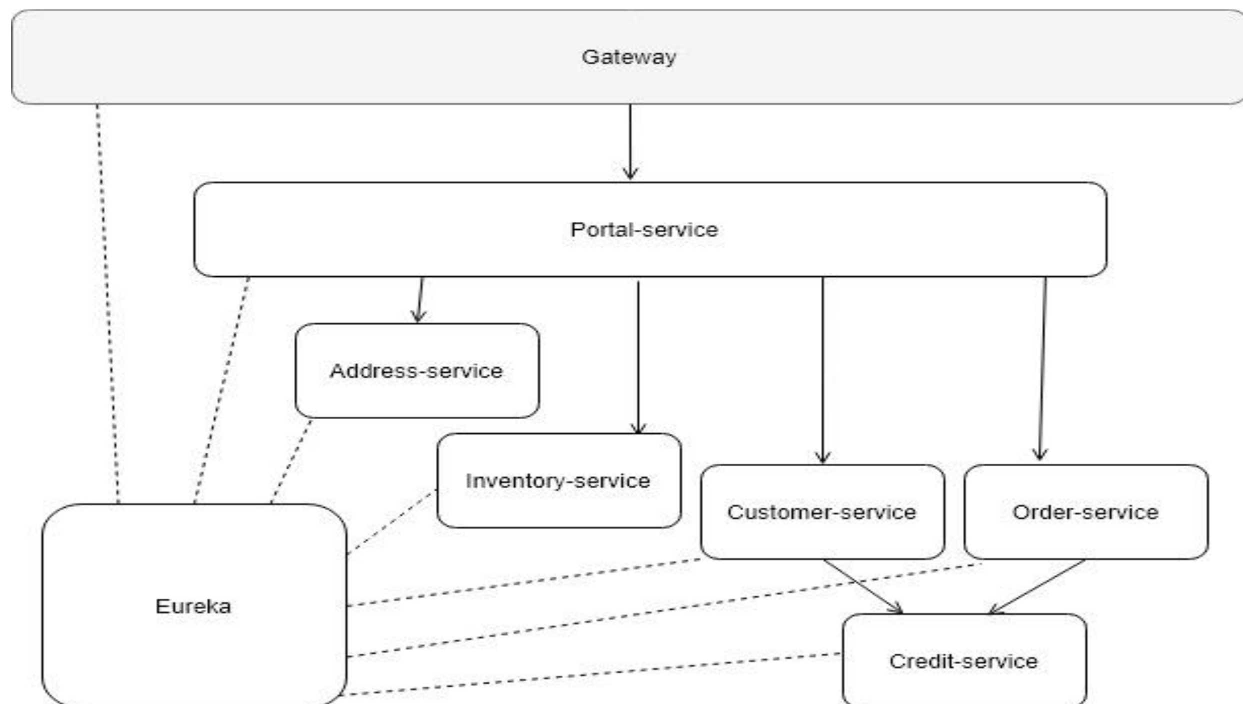

Figure 12. Exemplary microservice architecture

Address-service

Technology stack

- Java 8, IntelliJ IDE, Maven as Development Environment
- Spring-boot and Cloud as application framework
- Eureka as Service registry Server
- MSSQL server as data storage
- Docker to run MSSQL server

The address service is a RESTful microservice which exposes five end points. It is built using Java Spring boot framework and it uses MSSQL server running one Docker to store data. Address-service has five end points as described in Table 4.

Table 4. Address service endpoints

End point	Purpose
GET /address/id	Read address related to id
POST /address/add	Add new address
PUT /update/id	Update address related to id
PATCH /update/id	Update specific field of address related to id
Delete /address/id	Delete address related to id

Figure 13 shows the main application of the address service. The @EnableDiscoveryClient annotation will make the address service to be located as a client service and add to service discovery register automatically when address service instances up and running. Rest template bean is created and enabled as client-side load balancer using @LoadBalanced annotation.

```
@SpringBootApplication
@EnableDiscoveryClient
public class AddressServiceApplication {

   public static void main(String[] args) {
      SpringApplication.run(AddressServiceApplication.class, args);
   }

   @LoadBalanced
   @Bean
   RestTemplate restTemplate() {
      return new RestTemplate();
   }
}
```

Figure 13. Address service main application

Figure 14 describes service name and ports it is listening, service discovery URL which is running in the localhost, tracing system property set to trace all requests and URL where the traces will be send and the last property set is database property.

```
//service specific property
server.port=8070
spring.application.name=address-service

//service discovery property
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

// tracing system property
spring.sleuth.sampler.probability=100
spring.zipkin.baseUrl= http://localhost:9411/

// database prperty
spring.datasource.url = jdbc:sqlserver://localhost;databaseName=addressdb
spring.datasource.username = SA
spring.datasource.password = <STRONGPASSWORD>
spring.datasource.driver-class-name= com.microsoft.sqlserver.jdbc.SQLServerDriver
spring.jpa.show-sql= true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.hibernate.physical_naming_strategy =
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.SQLServer2012Dialect
spring.jpa.hibernate.connection.charSet = UTF-8
spring.jpa.properties.hibernate.show_sql = true
spring.jpa.properties.hibernate.format_sql = false
spring.jpa.properties.hibernate.default_schema = dbo
```

Figure 14. Address service application property

Eureka server

Microservice architecture involves one or more service instances. Every service instance has a unique name that is used to resolve its location. To have a unique name is so important as service will be discoverable, and it make services independent from infrastructure they are running on. This can be achieved by running the service instance on virtual machines or containers, which assign dynamic address (IP). On the other hand, service consumers need to know the exact address of each services it consumes.

There are two ways microservices to be aware of the services it requires. First one is to hard wire services which requires one another in the configuration file. That is to store the exact address and other related information in a configuration file. If the location of one the service change it needs to be updated in the configuration file so that other consumer services can find the new instance of microservice. This approach is not scalable and difficult to maintain. The second alternative is to have registry service that is capable of collect information about services, register and deregister service instances when services boot up or down. In this approach when a service needs to consume other service, first it asks registry service about the availability and whereabout of the service it want to consume.

In this project, Netflix Eureka service registry is setup and later client services are built which will register itself with the registry and uses it to resolve its own host. Figure 15 shows the service discovery service main class. The @EnableEurekaServer will enable the service to be service discovery eureka server and all other service which are set as client will register when instantiated.

```java
package com.danimeko.spring.tracing.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

   public static void main(String[] args) {
      SpringApplication.run(EurekaApplication.class, args);
   }
}
```

Figure 15. Service discovery Eureka service

Figure 16 shows the dependency need to be added to service discovery service.

```xml
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Figure 16. Eureka service dependency

Figure 17 describes service discovery service property which set port it listens, and it is set not to register itself as service to the service discovery.

```
// Port service discovery run
 server.port= 8761

// Self-registering disabled
 eureka.client.registerWithEureka = false
 eureka.client.fetchRegistry = false

 eureka.client.server.waitTimeInMsWhenSyncEmpty = 0
```

Figure 17. Eureka service property

Figure 18 shows service register Eureka running on localhost listening to port 8762. There listed seven services including gateway-service. It shows status and address of service instances registered with service discovery Eureka.

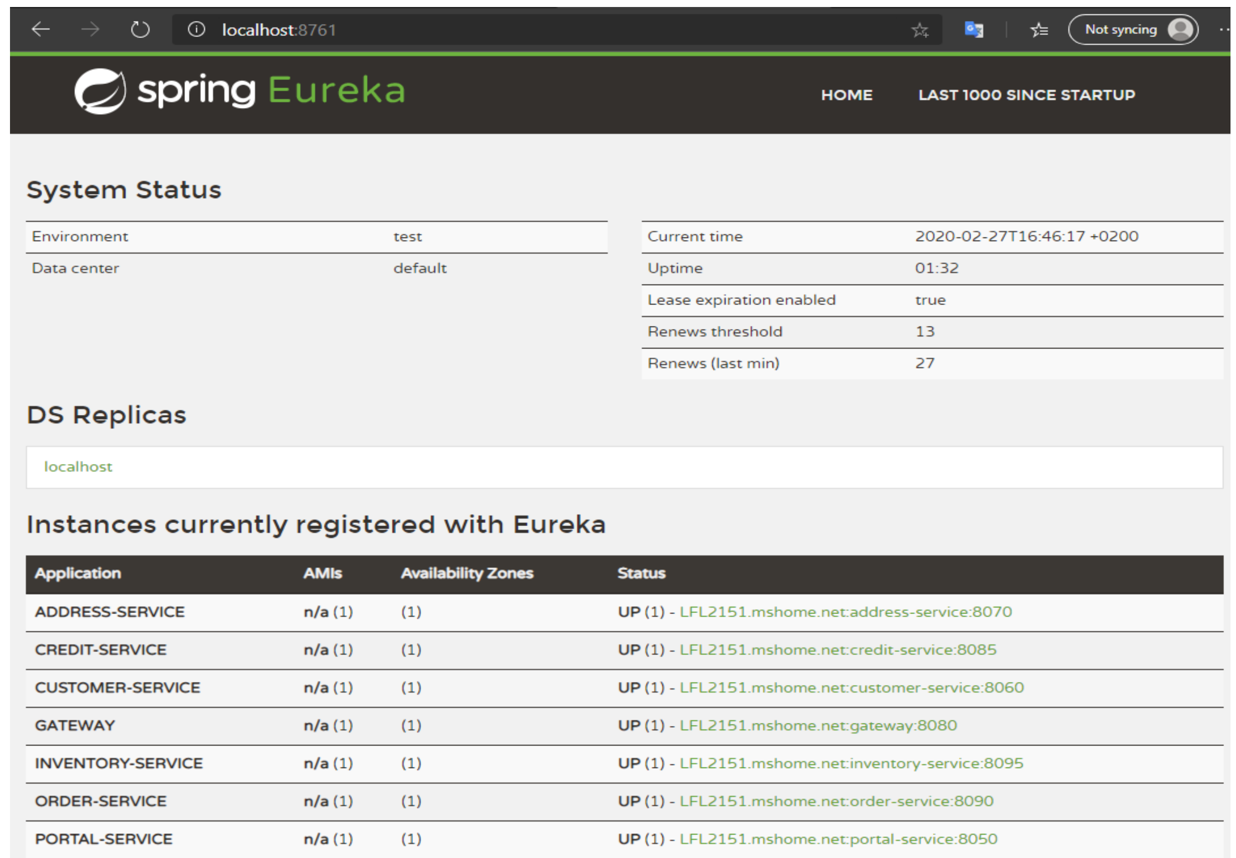

Figure 18. Registry service running, and service are registering

Portal-service

Portal-service is a spring cloud gateway microservice. It is a common entry point to access other microservices. When a request comes portal service will check if there is any service is registered in the service discovery which is in Eureka. If there is, it will get the address from Eureka and pass the request. This mean even though the services are running on different address, client service only need to know service name and directing request to portal-service it will route the request to the appropriate service in question. Figure 19 shows spring cloud gateway dependency for maven project.

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Figure 19. Spring cloud gateway dependency

For microservices to register itself to the service register it need to have @EnableDiscoveryClient annotation. Figure 20 shows @GatewayApplication has @EnableDiscoveryClient annotation that will itself-register to the service discovery.

```
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {

   public static void main(String[] args) {
      SpringApplication.run(GatewayApplication.class, args);
   }
}
```

Figure 20. Gateway application

Figure 21 describes common properties with other services in this exemplary microservice and some which are specific to it. For all microservice to be able to register to service discovery service, it is necessary to set the service discovery URL.

```
// port portal-service run
server.port = 8080
spring.application.name = gateway

// registering to service discovery
eureka.client.serviceUrl.defaultZone = http://localhost:8761/eureka/

spring.cloud.gateway.discovery.locator.lowerCaseServiceId = true
spring.cloud.gateway.discovery.locator.enabled = true

// tracing property
spring.sleuth.sampler.probability = 100
spring.zipkin.baseUrl= http://localhost:9411/
```

Figure 21. Gateway application property

In the tracing property, when spring.sleuth.sampler.probability set to 100, it means that all request coming to the service will be tracing. This can have significant effect on the application performance as those are directly proportional. Every microservice need to set the URL where to send the collected traces and spring.zipkin.baseUrl is the property set to http://localhost:9411.

4.2 Adding Instrumentation Library

Brave is a distributed tracing instrumentation library. Brave typically intercepts requests to gather timing data, correlate and propagate trace contexts. Spring cloud sleuth implements a distributed tracing system for Spring Cloud, and it also uses Brave (Brave. 2020). Spring cloud sleuth automatically instrument Spring applications.

The instrumentation is added by using a variety of technologies according to the stack that is available. For example, for a servlet web application, a Filter is used, and, for Spring Integration, ChannelInterceptors is used.

To trace every request end to end, every microservice need to have tracing library. Spring Cloud Sleuth need to be included in dependency. Figure 22 shows how the spring cloud sleuth dependency is added in maven project in the pom.xml file.

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Figure 22. Adding Spring cloud sleuth library in maven pom.xml file

The other feature Spring cloud sleuth have is to add [service-name, traceId, SpanId, exportable] entries to mapped diagnostic context (MDC) to the application log aggregator. Figure 23 shows "address-service" microservice with trace id, span id and exportable are added to MDC.

```
2020-02-26 17:08:58.227 DEBUG [address-service,263d95024b482b81,6475eb747d3dac6b,true]
2020-02-26 17:08:58.234 DEBUG [address-service,263d95024b482b81,6475eb747d3dac6b,true]
2020-02-26 17:08:58.388 DEBUG [address-service,263d95024b482b81,6475eb747d3dac6b,true]
```

Figure 23. Instrumented library adding information about the service running on MDC

4.3 Distributed Tracing with Zipkin and Jaeger

Zipkin is one of the distributed tracing systems that is compatible with openTracing standards. For these exemplary microservices, Zipkin will be tested running from docker. All exemplary microservices have property spring.zipkin.baseUrl set to http://localhost:9411 which is the address where the spans are sent. The spring.sleuth.sampler.probability property is the setting that set the sampling probability , setting to 100 means all incoming request are traced. The "spring-cloud-starter-zipkin" dependency will send spans to Zipkin over HTTP. The exemplary microservices are configured in a way that all service requests are passing through gateway-service. To send request to any of the service the format is the same except the service name and parameters.

Jaeger is distributed tracing system developed and open-sourced by Uber. Jaeger is like Zipkin but have different implementation. The exemplary microservices can send traces, which are the same as in Zipkin, then collected, analysed and displayed in Jaeger-UI component. The following command shown in Figure 24, run Jaeger in Docker and accept the same traces sent to Zipkin-collector. Jaeger-UI accessed from http://localhost:16686.

```
$ docker run -d --name jaeger \
 -e COLLECTOR_ZIPKIN_HTTP_PORT=9411 \
 -p 5775:5775/udp \
 -p 6831:6831/udp \
 -p 6832:6832/udp \
 -p 5778:5778 \
 -p 16686:16686 \
 -p 14268:14268 \
 -p 14250:14250 \
 -p 9411:9411 \
 jaegertracing/all-in-one:1.8
```

Figure 24. Running Jaeger from docker

When the following request shown on Figure 25 executed, Appendix 17 shows the type of JSON formatted trace sent to Zipkin and Jaeger.

```
curl --location --request POST 'http://localhost:8080/address-service/address/add/' \
--header 'Content-Type: application/json' \
--data-raw '{
        "streetname" :  "Närhitie",
        "postcode" : "99300",
        "city" : "Lapland",
        "country" : "Finland"
}'
```

Figure 25. POST request to address-service

The same JSON can be analysed and displayed in either Zipkin-UI or Jaeger-UI. Figure 27 and 28 shows how it is graphically displayed in Zipkin-UI and Jaeger-UI respectively.

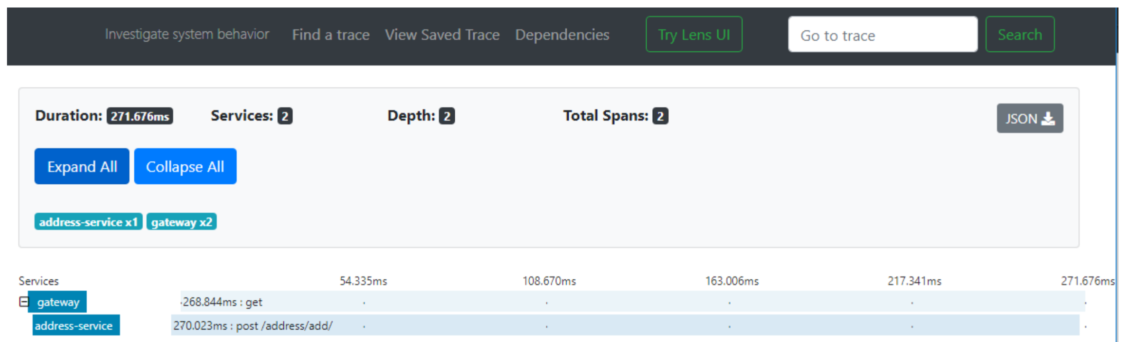

Figure 26. Zipkin-UI representation of a trace from Address-service

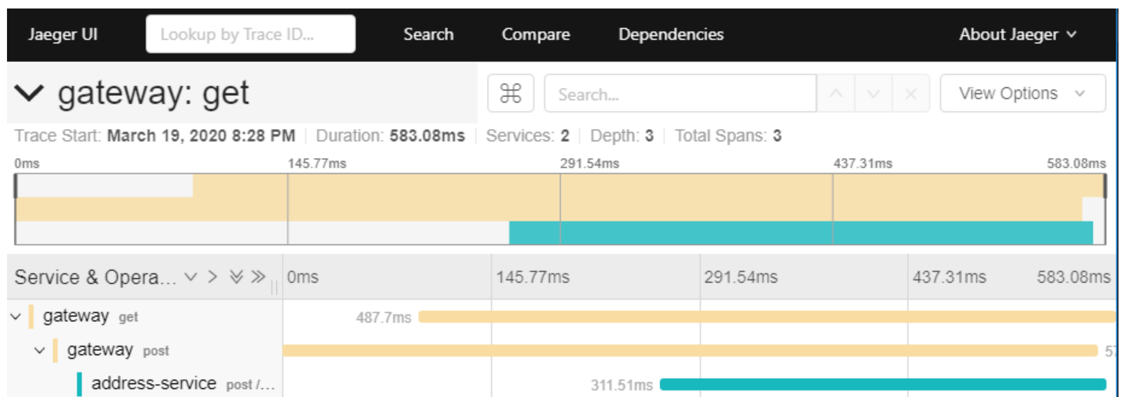

Figure 27. Jaeger-UI representation of a trace from Address-service

The following Figures 29 and 30 show spans and all related information collected in spans is represented in Zipkin.

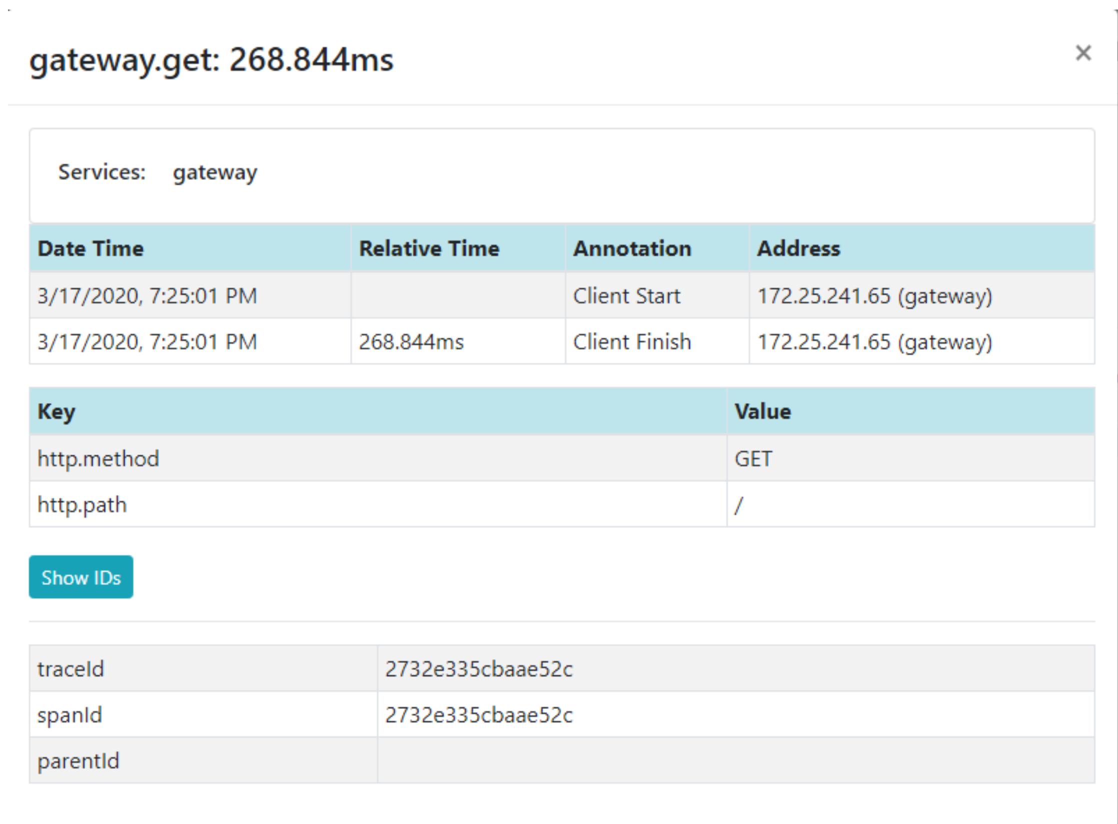

Figure 28. Zipkin-UI span representation from gateway-service

In Figure 29, the parentId is blank as gateway-service is where the tracing starts. The traceId = "2732e335cbaae52c" will serve as the parent id for the next span. Figure 30 describes address-service got the post request and took 270.023ms execution time.

address-service.post /address/add/: 270.023ms ×

Services: address-service,gateway

Date Time	Relative Time	Annotation	Address
3/17/2020, 7:25:01 PM	-2832µs	Client Start	172.25.241.65 (gateway)
3/17/2020, 7:25:01 PM	3.319ms	Server Start	172.25.241.65 (address-service)
3/17/2020, 7:25:01 PM	267.191ms	Client Finish	172.25.241.65 (gateway)
3/17/2020, 7:25:01 PM	268.088ms	Server Finish	172.25.241.65 (address-service)

Key	Value
http.method	POST
http.path	/address/add/
mvc.controller.class	AddressController
mvc.controller.method	address
Client Address	[::1]

Show IDs

traceId	2732e335cbaae52c
spanId	bef1bf5f0c6269e0
parentId	2732e335cbaae52c

Figure 29. Zipkin-UI detail presentation of a trace

Jaeger-UI has also detailed presentation of a trace. Figure 31 describes a trace for POST request sent to address-service pass through gateway-service.

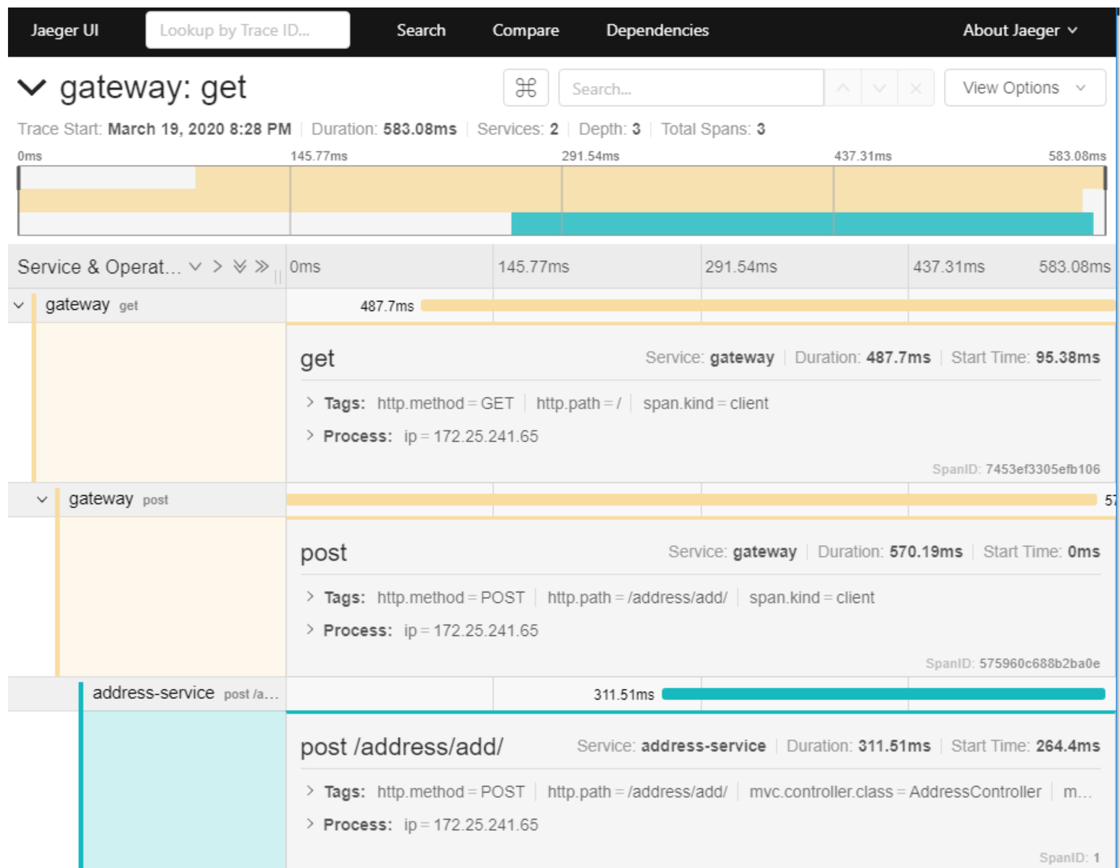

Figure 30. Jaeger-UI detail presentation of a trace

Tracing more than one microservice, which is tracing distributed microservice, is a combination spans from each of microservice as a request progress from one to the next and present it a way that that answer what has happened where and how long the execution took in each microservice. In the exemplary microservice a request to portal-service will call multiple of microservice. The following request the portal-service with the parameter of id = 3. Figure 28 shows more than one microservice graphically presented in Zipkin-UI.

curl --location --request GET 'http://localhost:8080/portal-service/fullDetails/3'

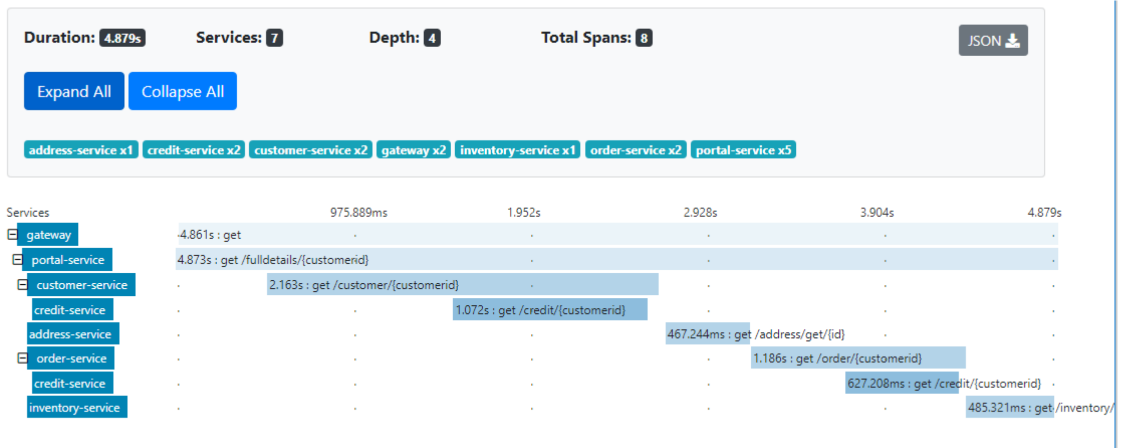

Figure 31. Seven microservices are presented in Zipkin-UI

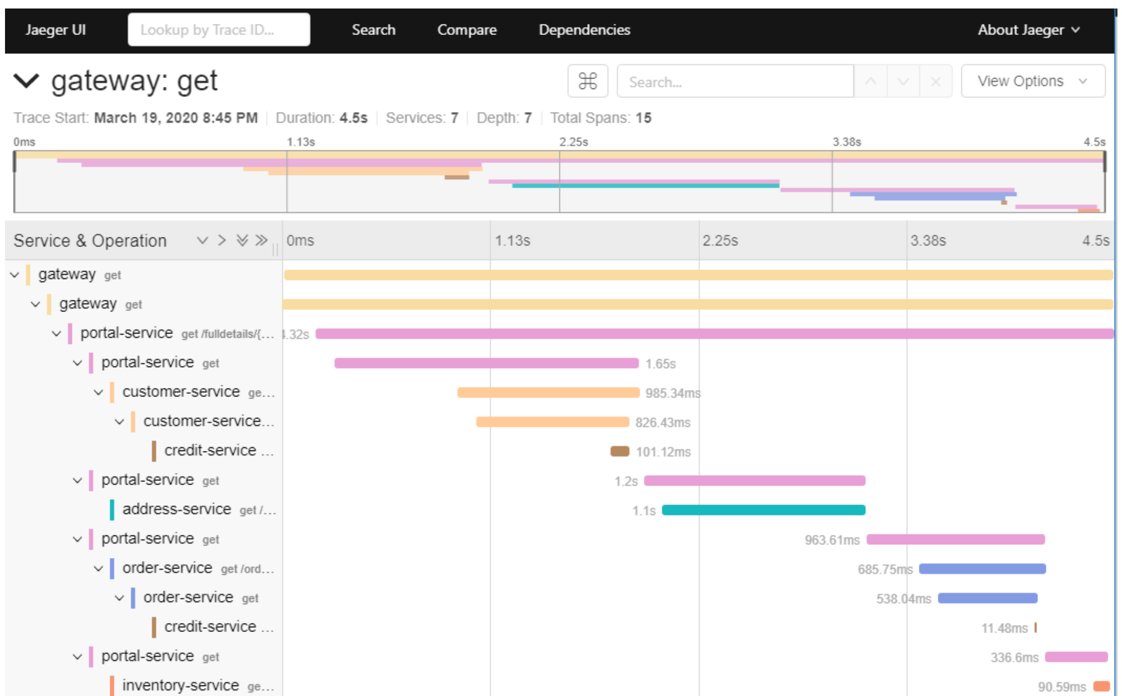

Figure 32. Seven microservices are presented in Jaeger-UI

Jaeger-UI has dependencies page that shows a graph of services traced by Jaeger and connections between them. Since the Jaeger is running in localhost and data are stored in memory, it is possible to generate a dependency graph as collected tracing data will not be large. In production environment this need separate task as it need to process all data stored in the database. Figure 34 shows dependency graph for the exemplary microservices.

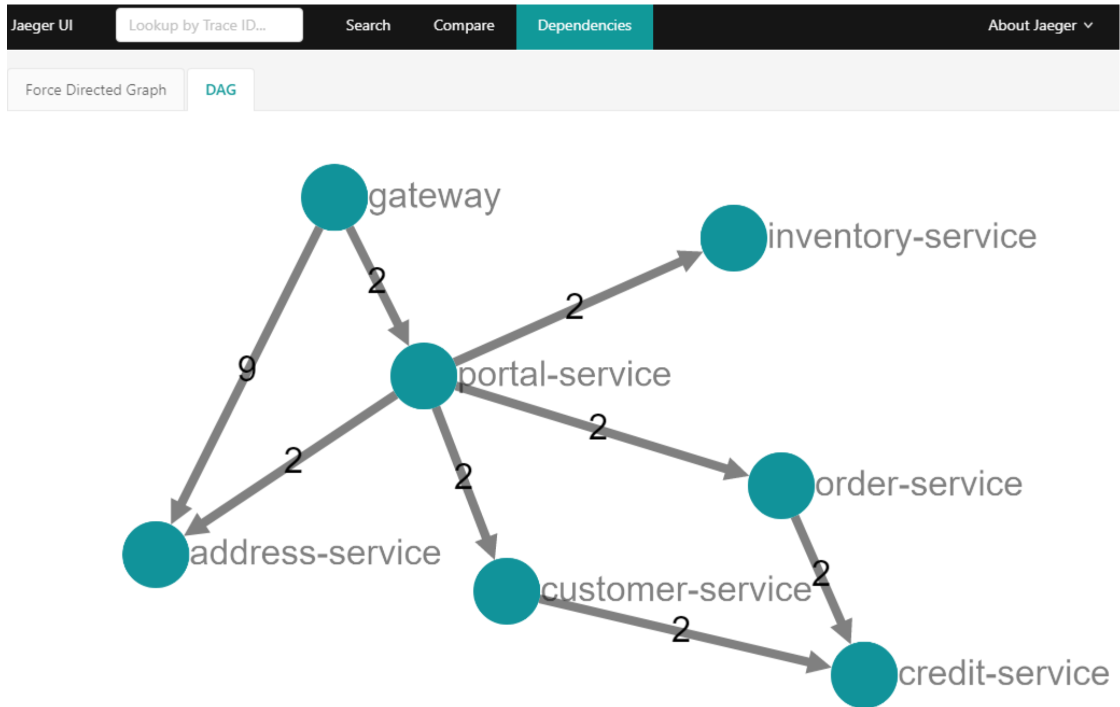

Figure 33. Dependency graph

4.4 Visualizing Errors

When one of the microservices in the path of the request fail for some reason, distributed tracking system can point where the failure happened. To demonstrate error visualization using Zipkin-UI or Jaeger-UI, there will be no instance of customer-service running in the exemplary microservice. Figure 35 and 36 shows the graphical representation when a microservice failure using Zipkin-UI and Jaeger-UI respectively. In Figure 35, the service marked in red is where the system failure happened and it is marked twice which means that, the failed service has been called twice for normal operation.

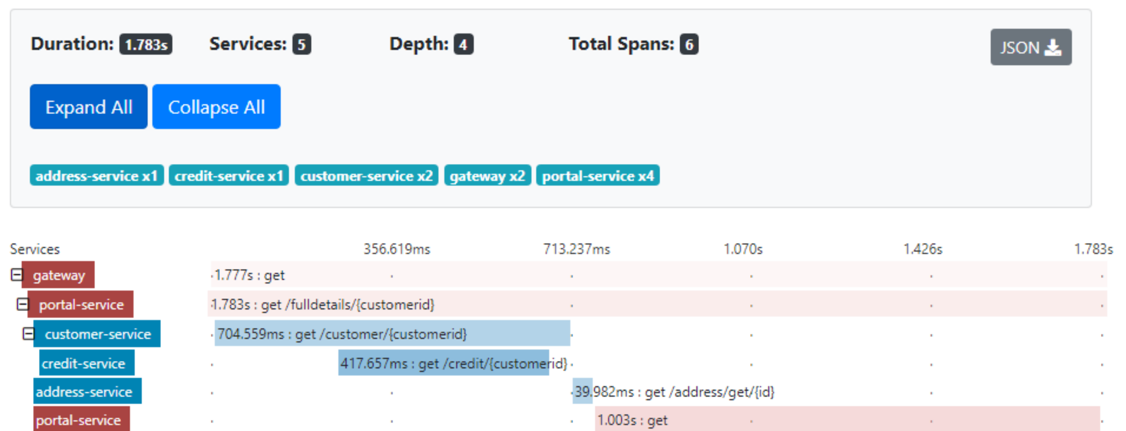

Figure 34. Zipkin-UI representation of a service failure in portal service

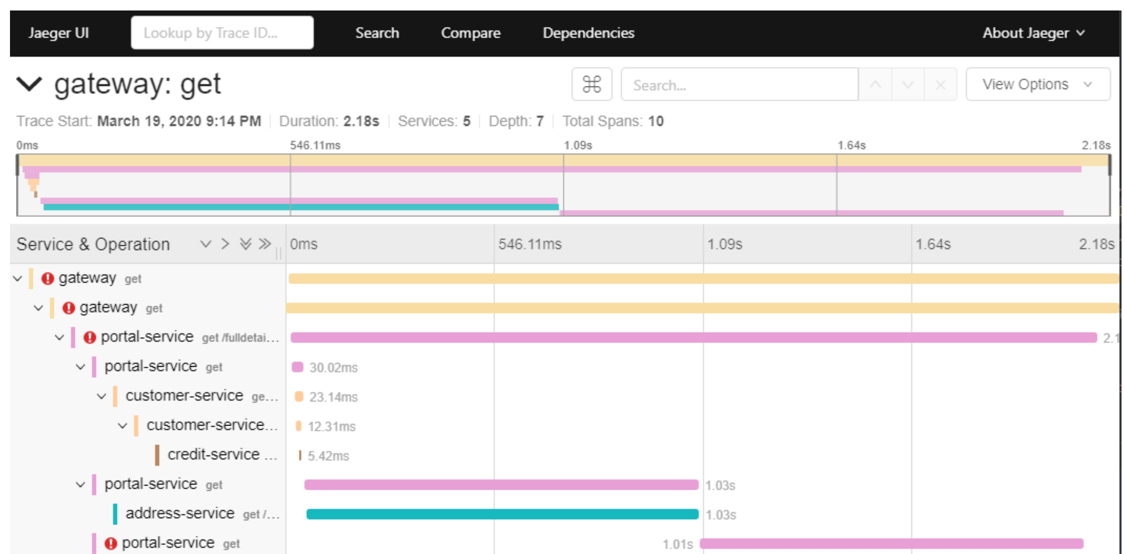

Figure 35. Jaeger-UI representation of a service failure in portal service

4.5 Detecting Latency

Detecting latency in microservice architecture is a big task as it directly affects user experience. Finding which service is taking time in microservice architecture for a request can be done in distributed tracing system. The exemplary microservice tested by introducing delay in address-service. Figure 37 shows the address-service AddressController component introduced delay.

```
@RequestMapping(value = "/get/{id}")
Address getAddress(@PathVariable(name = "id") Integer id ) {

// delay in seconds
    try {

        TimeUnit.SECONDS.sleep(12);

    } catch (InterruptedException ie) {

        Thread.currentThread().interrupt();
    }

    return  service.getById(id);
}
```

Figure 36. Inserting delay in seconds

The introduced delay will increase the execution time for all subsequent operation and finally the effect will also be reflected on the overall execution time. Figure 38 shows the Jaeger-UI representation of the execution time per service.

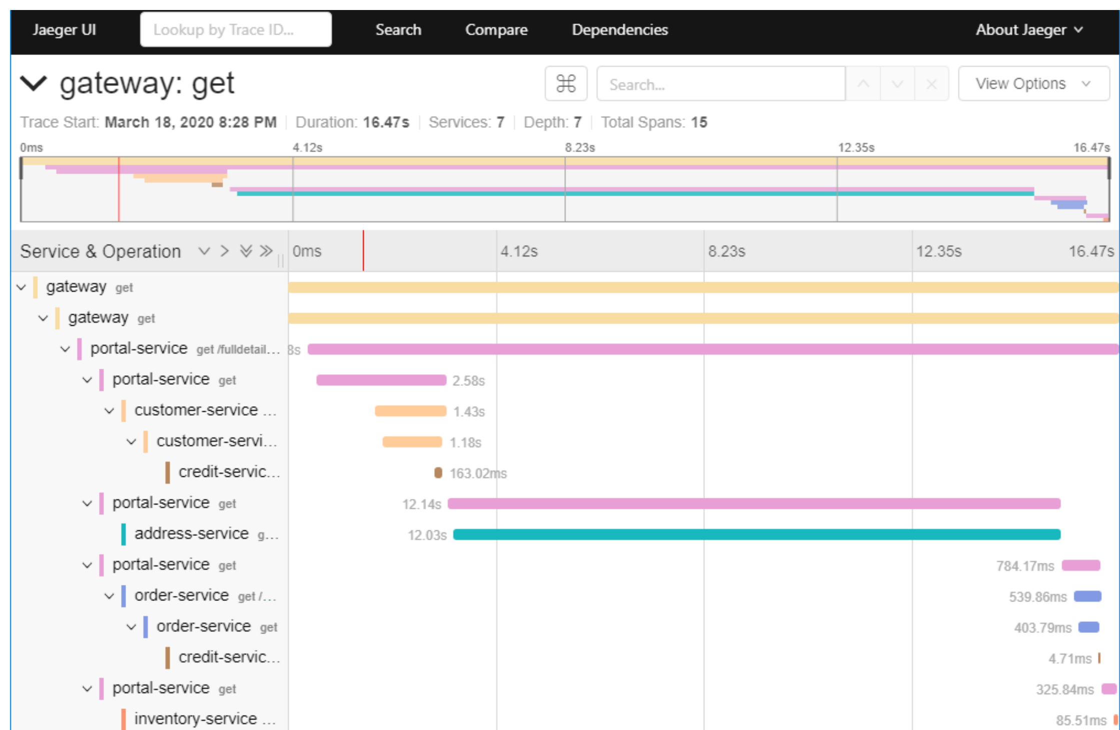

Figure 37. Jaeger-UI shows execution time per service

The above figure indicates that execution time coloured in green took 76.06% of total execution time and from the side menu indicates that, portal-service call address-service that took 12.03 seconds. This means there is a need to optimize or some maintenance work is needed at address-service.

5 Results

Research questions

The research questions presented in chapter 1 were the following:

1. How to trace a request in distributed microservices applications?

2. How to instrument a distributed microservices application?

3. How to find latency in microservices using a distributed tracing system?

4. What are the available open-source distributed microservice tracing systems and criteria to choose one?

A request coming to a microservice or any system can be traced using an instrumentation library. Instrumentation library is a library specific to programming language and framework. It resides in the client's code and records metadata of what has happened with a request as it passes through each set of instruction in microservice. The collected metadata will later be pushed into a collector, a storage of metadata, as the request progresses to the next microservice. Those microservices can sometimes be in different physical location, and they can also can be written in a different programming language, so there is some information attached to the request header that later can be used to relate the collected metadata and form the complete history of a request's journey. When the whole execution ends for a request, complete metadata is collected. This data will be analyzed and presented to understand and answer many questions about all microservices it passes.

The two commonly used open-tracing systems can be used to analyze the metadata and present different aspect of microservice for instance, execution time in each service and dependency graph present which service is depend on which service.

Generally, both Zipkin and Jaeger distributed tracing systems trace, collect and analyze requests. However, there are criteria that one needs to consider before taking one of them into use.

Official Language and Library support

Table 5 lists language and libraries officially supported by Zipkin and Jaeger. They both have clear instruction and documentation how to write custom instrumentation library if someone wants to write custom instrumentation library. Also, there are few alternatives instrumentation libraries for the officially supported languages as well from Zipkin and Jaeger community.

Table 5. Official language and Library support

Language	**Zipkin**	**Jaeger**
C#	yes	yes
C++	no	yes
Java	yes	yes
Go	yes	yes
Python	no	yes
JavaScript	yes	no
Nodejs	no	yes
Ruby	yes	no
PHP	yes	no
Scala	yes	no

Deployment

There are three ways to use Zipkin. Java, running the self-contained executable jar is running it from the source code and running from Docker.

Unlike Zipkin, which is isolated project, Jaeger is part of the CNCF (Cloud Native Computing Foundation) project or eco-system, which means its preferred deployment is Kubernetes. It is also possible to run Jaeger in Docker.

Community

Jaeger has a relatively small community comparing to community members since the time of the release of its open-source community in 2017. Since Zipkin has been released to the open-source community in 2012, it has gathered a large community of members and has instrumentation libraries for most languages. Table 6 shows a simple comparison between Zipkin and Jaeger from their online community in Gitter chat and number of stars in repos.

Table 6. number of Zipkin's and Jaeger's community and rating

	Number of community members in Gitter chat	Number of stars in GitHub
Zipkin	2030	12575
Jaeger	1082	10450

Zipkin has been around for a while and the adaptation of Jaeger by the Cloud native computing foundation makes it suitable for the cloud deployment. It works with containers, e.g. Kubernetes and supports OpenTracing and the ecosystem around CNCF.

Finally, one reason why Jaeger is getting good momentum in a very short time is its being a part of an CNCF echo system, and Jaeger is compatible with Zipkin’s API, meaning the Zipkin instrumentation libraries can be used with Jaeger’s collector.

6 Discussion

The research questions presented in Chapter 1 are the foundation of this research. Monolithic architecture for large backend is increasingly rare. It resulted an increase in adaptation of distributed microservices; which carry out small and specific tasks and communicate with each other. The research focuses on solving the difficulty in tracing and analyzing requests in a distributed microservice environment. Locating problems can be very difficult as a request may touch many services. In some situations, there is a need for different teams to find time and sit together to find out a problem in a distributed microservice. Distributed tracing system tackles those problems in a very practical way and finding a problem or debugging relatively easy and straight forward.

Company X, which assigned this study, has a few legacy codes and at the moment it is shifting to microservice and cloud implementation. An exemplary microservice architecture was built which resembles the microservice in Company X. The exemplary microservice is instrumented with tracing library Spring cloud Sleuth, which implements a distributed tracing system for Spring Cloud, and it also uses Brave.

A tracing library collects metadata to every request in each service. The collected data can be what operation has been done, how long it took to carry it out, what other service it communicates with, and how long the communicated service took. These collected data passed to storage for analysis.

In the research Zipkin and Jaeger were considered for analysis and visualization of the collected data. Both are inspired by Google paper (Sigelman et al. 2010) and have a wide support for different programming languages and frameworks. Both Zipkin and Jaeger are also open-sourced and have a large community. Jaeger is part of CNCF, which makes it suitable for cloud deployment.

The research was able to demonstrate how to instrument tracing library on microservice built on Spring framework, and how to collect, analyze and visualize traces. It also answered to the question what service the request passes through, latency for an individual service and for a whole request journey.

The research used an exemplary microservice which was based on Java Spring Framework. In most cases a service used more than one microservice and that microservice may not have been built in the same programming language. Theoretically, even if services are built in a different programming language, tracing information about the parent trace can pass from one service to the next service. There were no such cases to test in this research.

I found the research gratifying, because finding latency and a source of bugs in microservice architecture is a very tedious and time-consuming task. For many companies moving towards microservice architecture, the request tracing system must be a matter to consider, especially when the number of services increase. Furthermore, I have learned much when I built the exemplary microservice architecture. For instance, service discovery, service gateway, and the six services are all built by me.

7 Further research and development

Frontend to backend

These days most of the frontends are static; it is also possible to add tracing to the frontend. This thesis covers the backend only, which is where the tracing starts after it has passed the frontend. In my opinion, if the frontend were also traced, it would be possible to show a full visualization of the request route.

Testing in production environment

In the thesis, exemplary microservices built using Java Spring framework were used. These microservices have basic functionality and single operations. Furthermore, all of them are running in the same local development environment and location. MSSQL (Microsoft SQL server) database using in the exemplary microservice is also running from a local environment in Docker. Even if service discovery and load balancing are used, these exemplary microservices lack the nature of microservices; in reality, microservices can be built using different programming languages and teams. Even if a microservice does one task, it runs complex multiple operations, running in different environment and physical location.

References

Architecture. 2020. Page on Zipkins' website. Accessed on 26 January 2020. Retrieved from https://zipkin.io/pages/architecture.html

Architecture. 2020. Page on Jaeger tracing website. Accessed on 19 January 2020. Retrieved from https://www.jaegertracing.io/docs/1.16/architecture/

Brave. 2020. Page on Braves' github page. Accessed on 12 March 2020. Retrieved from https://github.com/openzipkin/brave

Distributed Tracing for Microservices. Jaeger vs. Zipkin. 2019. Page on Bizetys' website. Accessed on 02 February 2020. Retrieved from https://www.bizety.com/2019/01/14/distributed-tracing-for-microservices-jaeger-vs-zipkin/

Fowler, M. & Lewis, J. 2014. Microservices. Microservices - a definition of this new architectural term. Accessed on 05 September 2019. Retrieved from https://www.martinfowler.com/articles/microservices.html

George, F. 2013. MicroService Architecture. A Personal Journey of Discovery. Accessed 09 Feburary 2020. Retrieved from https://www.slideshare.net/fredgeorge/micro-service-architecure

Global Microservices Trends Report. 2018. Page on LightSteps' website. Accessed on 19 September 2019. Retrieved from https://go.lightstep.com/rs/260-KGM-472/images/global-microservices-trends-2018.pdf

Kharenko, A. 2015. Monolithic vs. Microservices Architecture. Accessed on 25 Septamber 2019. Retrieved from https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

Lewis, J. 2012. Microservices - Java, the Unix Way. Accessed 08 September 2019. Retrieved from http://2012.33degree.org/talk/show/67

Partition (database). 2019. Page on Wikipedia. Accessed on 26 September 2019. Retrieved from https://en.wikipedia.org/wiki/Partition_(database)

Sampling. 2020. Page on Jaeger tracing website. Accessed on 19 January 2020. Retrieved from https://www.jaegertracing.io/docs/1.14/sampling/

Shkuro, Y. 2019a. Embracing context propagation. Accessed on 20 January 2020. Retrieved from https://medium.com/jaegertracing/embracing-context-propagation-7100b9b6029a

Shkuro, Y. 2019b. Mastering Distributed Tracing. Accessed on 20 January 2020. Retrieved from https://www.packtpub.com/eu/networking-and-servers/mastering-distributed-tracing

Shkuro, Y. 2017. Evolving Distributed Tracing at Uber Engineering. Accessed on 01 February 2020. Retrieved from https://eng.uber.com/distributed-tracing/

Spans. 2020. Page on OpenTracings' website. Accessed on 10-02-2020. Retrieved from https://opentracing.io/docs/overview/spans/

Sigelman, B., Barroso, L., Burrows, M., Stemphenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C. 2010. Dapper. A Large-Scale Distributed Systems Tracing Infrastructure. Accessed on 26 September 2019. Retrieved from https://research.google.com/archive/papers/dapper-2010-1.pdf

Tracers and Instrumentation. 2020. Page on Zipkins' website. Accessed on 26 January 2020. Retrieved from https://zipkin.io/pages/tracers_instrumentation.html

Zipkin API. 2020. Page on Zipkins' website. Accessed on 15 January 2020. Retrieved from https://zipkin.io/zipkin-api/

8 Appendices

Appendix 1. The exemplary microservice structure and pattern

Appendix 2. Address-service address entity component

```java
package com.danimeko.spring.tracing.entities;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table( name = "address"  )
@JsonIgnoreProperties( { "hibernateLazyInitializer", "handler" } )
public class Address {

    @Id
    @GeneratedValue( strategy = GenerationType.AUTO )
    private Integer id;

    @Column( name = "streetname" )
    private String streetname;

    @Column( name = "postcode" )
    private String postcode;

    @Column( name = "city" )
    private String city;

    @Column( name = "country" )
    private String country;

    public Integer getId() { return id;}

    public void setId(Integer id) { this.id = id; }

    public String getCity() { return city; }

    public void setCity(String city) { this.city = city; }

    public String getCountry() { return country; }

    public void setCountry(String country) { this.country = country; }

    public String getStreetname() { return streetname; }

    public void setStreetname(String streetname) { this.streetname = streetname; }

    public String getPostcode() { return postcode; }

    public void setPostcode(String postcode) {  this.postcode = postcode; }

}
```

Appendix 3. Address-service controller component // the appendix on the same page

```java
package com.danimeko.spring.tracing.controller;

import com.danimeko.spring.tracing.entities.Address;
import com.danimeko.spring.tracing.service.AddressService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
.*;
import java.util.concurrent.TimeUnit;

@RestController
@RequestMapping(value = "/address")
public class AddressController {

    @Autowired
    private AddressService service;

    @RequestMapping(value = "/add")
    Address address(@RequestBody Address address){
        return service.save(address);
    }

    @RequestMapping(value = "/update/{id}")
    Address updateAddress(@RequestBody Address address,
                          @PathVariable(name = "id") Integer id){

        Address oldAddress = service.getById(id);

        if( address.getStreetname() != null )
        oldAddress.setStreetname(address.getStreetname());
        if( address.getCity() != null )
        oldAddress.setCity(address.getCity());
        if( address.getPostcode() != null )
        oldAddress.setPostcode(address.getPostcode());
        if( address.getCountry() != null )
        oldAddress.setCountry(address.getCountry());

       return service.save(oldAddress);
    }

    @RequestMapping(value = "/get/{id}")
    Address getAddress(@PathVariable(name = "id") Integer id ) {

        return  service.getById(id);
    }

    @RequestMapping(value = "/delete/{id}")
    String deleteAddress(@PathVariable(name = "id", required = true) Integer id ){

         service.delete(id);
        return "Address of id = "+ id + " deleted";
    }
}
```

Appendix 4. Address-service service component

```
package com.danimeko.spring.tracing.service;

import com.danimeko.spring.tracing.entities.Address;
import com.danimeko.spring.tracing.repository.AddressRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class AddressService {
    private final AddressRepository addressRepository;

    @Autowired
    public AddressService(AddressRepository addressRepository) {
        this.addressRepository = addressRepository;
    }

    public List<Address> getAll(){
        return addressRepository.findAll();
    }

    public Address save(Address address){
        return addressRepository.save(address);
    }

    public Address getById(Integer id){
        return addressRepository.getOne(id);
    }

    public void delete(Integer id){ addressRepository.deleteById(id);}
}
```

Appendix 5. Address-service repository component

```
package com.danimeko.spring.tracing.repository;

import com.danimeko.spring.tracing.entities.Address;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AddressRepository extends JpaRepository<Address,Integer> {
}
```

Appendix 6. Address-service main component

```java
package com.danimeko.spring.tracing;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableDiscoveryClient
public class AddressServiceApplication {

   public static void main(String[] args) {
      SpringApplication.run(AddressServiceApplication.class, args);
   }

   @LoadBalanced
   @Bean
   RestTemplate restTemplate() {
      return new RestTemplate();
   }
}
```

Appendix 7. Address-service application.property

```properties
server.port=8070
spring.application.name=address-service
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

spring.sleuth.sampler.probability=100
spring.zipkin.baseUrl= http://localhost:9411/

spring.datasource.url = jdbc:sqlserver://localhost;databaseName=addressdb
spring.datasource.username = SA
spring.datasource.password = <password>
spring.datasource.driver-class-name= com.microsoft.sqlserver.jdbc.SQLServerDriver

spring.jpa.show-sql= true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.hibernate.physical_naming_strategy =
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.SQLServer2012Dialect
spring.jpa.hibernate.connection.charSet = UTF-8
spring.jpa.properties.hibernate.show_sql = true
spring.jpa.properties.hibernate.format_sql = false
spring.jpa.properties.hibernate.default_schema = dbo
```

Appendix 8. Address-service pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>com.danimeko.spring.micorservice</groupId>
      <artifactId>tracing-microservice</artifactId>
      <version>0.0.1-SNAPSHOT</version>
   </parent>
   <groupId>com.danimeko.spring.tracing</groupId>
   <artifactId>address-service</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <name>address-service</name>
   <description>Demo project for Spring Boot tracing</description>

   <properties>
      <java.version>1.8</java.version>
      <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>

      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
      </dependency>

      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-sleuth</artifactId>
      </dependency>

      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-zipkin</artifactId>
      </dependency>

      <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-
boot-starter-jdbc -->
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-jdbc</artifactId>

      </dependency>

      <!-- https://mvnrepository.com/artifact/org.springframework.data/spring-
data-commons -->
      <dependency>
         <groupId>org.springframework.data</groupId>
         <artifactId>spring-data-commons</artifactId>
      </dependency>
```

```
      <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-
boot-starter-data-jpa -->
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-data-jpa</artifactId>

      </dependency>

      <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-core</artifactId>
         <version>5.2.15.Final</version>
      </dependency>

      <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-
entitymanager -->
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-entitymanager</artifactId>
         <version>5.2.15.Final</version>
      </dependency>

      <!-- https://mvnrepository.com/artifact/com.microsoft.sqlserver/mssql-jdbc
      <dependency>
         <groupId>com.microsoft.sqlserver</groupId>
         <artifactId>mssql-jdbc</artifactId>
         <version>7.4.1.jre11</version>
      </dependency>-->

      <dependency>
         <groupId>javax.xml.bind</groupId>
         <artifactId>jaxb-api</artifactId>
         <version>2.3.0</version>
      </dependency>

      <dependency>
         <groupId>com.microsoft.sqlserver</groupId>
         <artifactId>mssql-jdbc</artifactId>
         <version>6.1.0.jre8</version>
      </dependency>

      <!-- https://mvnrepository.com/artifact/org.javassist/javassist -->
      <dependency>
         <groupId>org.javassist</groupId>
         <artifactId>javassist</artifactId>
         <version>3.25.0-GA</version>
      </dependency>

      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <scope>test</scope>
      </dependency>
   </dependencies>
```

```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Appendix 9. Parent project pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.danimeko.spring.micorservice</groupId>
   <artifactId>tracing-microservice</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>pom</packaging>

   <name>tracing</name>
   <description>Demo project for spring microservices tracing</description>

   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.1.6.RELEASE</version>
      <relativePath/> <!-- lookup parent from repository -->
   </parent>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
      <java.version>1.8</java.version>
      <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
   </properties>

   <dependencyManagement>
     <dependencies>
         <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>

   <modules>
       <module>eureka</module>
      <module>gateway</module>
      <module>address-service</module>
      <module>portal-service</module>
      <module>customer-service</module>
      <module>order-service</module>
      <module>credit-service</module>
      <module>inventory-service</module>
   </modules>


</project>
```

Appendix 10. Eureka service discovery microservice pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>com.danimeko.spring.micorservice</groupId>
      <artifactId>tracing-microservice</artifactId>
      <version>0.0.1-SNAPSHOT</version>
   </parent>
   <groupId>com.danimeko.spring.tracing</groupId>
   <artifactId>eureka</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <name>eureka</name>
   <description>Eureka service register</description>

   <properties>
      <java.version>1.8</java.version>
      <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
      </dependency>
   </dependencies>

   <dependencyManagement>
      <dependencies>
         <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>

   <build>
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
         </plugin>
      </plugins>
   </build>

</project>
```

Appendix 11. Eureka service discovery microservice main component

```
package com.danimeko.spring.tracing.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
   public static void main(String[] args) {
      SpringApplication.run(EurekaApplication.class, args);
   }
}
```

Appendix 12. Eureka service discovery microservice application.property

```
server.port= 8761

eureka.client.registerWithEureka= false
eureka.client.fetchRegistry= false
eureka.client.server.waitTimeInMsWhenSyncEmpty=0
```

Appendix 13. Gateway-service main component

```
package com.danimeko.spring.tracing.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {

   public static void main(String[] args) {
      SpringApplication.run(GatewayApplication.class, args);
   }

}
```

Appendix 14. Gateway-service pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>com.danimeko.spring.micorservice</groupId>
      <artifactId>tracing-microservice</artifactId>
      <version>0.0.1-SNAPSHOT</version>
   </parent>
   <groupId>com.danimeko.spring.tracing</groupId>
   <artifactId>gateway</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <name>gateway</name>
   <description>Demo project for Spring Boot Tracing</description>

   <properties>
      <java.version>1.8</java.version>
      <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-gateway</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-sleuth</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-zipkin</artifactId>
      </dependency>

      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <scope>test</scope>
      </dependency>
   </dependencies>

   <dependencyManagement>
      <dependencies>
         <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>
```

```
    <build>
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
         </plugin>
      </plugins>
   </build>

</project>
```

Appendix 15. Gateway-service application.property

```
server.port=8080
spring.application.name=gateway
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true
spring.cloud.gateway.discovery.locator.enabled=true

spring.sleuth.sampler.probability=100
spring.zipkin.baseUrl= http://localhost:9411/
```

Appendix 16. Portal-service portalController component

```
package com.danimeko.spring.tracing.portalservice;

import com.danimeko.spring.tracing.portalservice.entities.Address;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class PortalController {

    private static Logger log = LoggerFactory.getLogger(PortalController.class);

    @Autowired
    RestTemplate restTemplate;

    @GetMapping(value = "/fullDetails/{customerId}")
    public String address(@PathVariable(name = "customerId", required = true)
Integer customerId){
        log.info("GET /fullDetails/"+customerId);

        String customerResponse=restTemplate.getForObject("http://customer-
service/customer/"+customerId,String.class);

        Address addressResponse=restTemplate.getForObject("http://address-
service/address/get/"+customerId, Address.class);

        String orderResponse=restTemplate.getForObject("http://order-
service/order/"+customerId,String.class);

        String inventoryResponse=restTemplate.getForObject("http://inventory-
service/inventory/",String.class);

        assert addressResponse != null;
        return customerResponse+ "<br>" +addressResponse.getCountry()+ "<br>"
+orderResponse +"<br>" + inventoryResponse + "<br>" ;
    }
}
```

Appendix 17. JSON formatted trace sent to Zipkin

[{
"traceId": "2732e335cbaae52c",
"parentId": "2732e335cbaae52c",
"id": "bef1bf5f0c6269e0",
"kind": "SERVER",
"name": "post /address/add/",
"timestamp": 1584465901541198,
"duration": 264769,
"localEndpoint": {
"serviceName": "address-service",
"ipv4": "172.25.241.65"
},
"remoteEndpoint": {
"ipv6": "::1"
},
"tags": {
"http.method": "POST",
"http.path": "/address/add/",
"mvc.controller.class": "AddressController",
"mvc.controller.method": "address"
},
"shared": true
},
{
"traceId": "2732e335cbaae52c",
"parentId": "2732e335cbaae52c",
"id": "bef1bf5f0c6269e0",
"kind": "CLIENT",
"name": "post",
"timestamp": 1584465901535047,
"duration": 270023,
"localEndpoint": {

"serviceName": "gateway",
"ipv4": "172.25.241.65"
},
"tags": {
"http.method": "POST",
"http.path": "/address/add/"
}
},
{
"traceId": "2732e335cbaae52c",
"id": "2732e335cbaae52c",
"kind": "CLIENT",
"name": "get",
"timestamp": 1584465901537879,
"duration": 268844,
"localEndpoint": {
"serviceName": "gateway",
"ipv4": "172.25.241.65"
},
"tags": {
"http.method": "GET",
"http.path": "/"
}
}]