Fei Xu, Yichen Lin *Research Course*

Fall | 2016

Technical Report

Microservices Tracking System

Team

Enclosed in this document is the technical report of the <Project Name> sponsored by <Sponsor Name>.

1. Introduction

2. Motivation

3. Related work

4. System design
    4.1 Billing Microservices system design
        4.1.1 Registration Service - Eureka
        4.1.2 Web Service
        4.1.3 Accounts Service
        4.1.4 e-Business Service
        4.1.5 Order Service
    4.2 Distributed Tracking
        4.2.1 Sleuth+exposed by REST APIs
        4.2.2 Sleuth Stream + RabbitMQ + Zipkin
        4.2.3 Sleuth Stream + RabbitMQ + Customized Customer
        4.2.3 Sleuth + Zipkin
    4.3 DB collection and Display
        4.3.1 DB collection and processing
            4.3.1.1 Data Structure used in Sleuth
            4.3.1.2 Design of the Collection Service
        4.3.2 D3 Display

5. System Implementation
    5.1 Billing microservices system implementation
        5.1.1 Services Registration - Eureka
        5.1.2 Accounts Service:
        5.1.3 Order Service
        5.1.4 Web Service
    5.2 Distributed Tracing
        5.2.1 Sleuth+exposed by UI
        5.2.2 Sleuth Stream+RabbitMQ+Zipkin
        5.2.3 Sleuth Stream + RabbitMQ + Custom Consumer
        5.2.4 Sleuth+Zipkin
    5.3 DB collection and Display

5.3.1DB collection and processing
5.3.2 D3 Display

6. Tutorial & Experiments & Analysis
6.1 Distributed Tracing
6.1.1 Sleuth(core) experiments
6.1.2 Sleuth Stream + RabbitMQ + Zipkin
6.1.3 Sleuth Stream + RabbitMQ + Customized Consumer
6.1.4 Sleuth + Zipkin Collector(Customized Consumer) + D3
a. Get the Billing microservices system start running
b. Configure local database
c. Get Zipkin service start running
d. Get the Collector-Service running
e. Get the D3 service start running

7.Conclusions and Thoughts

8. Contribution of each team member

Appendix:

References

1. Introduction

Microservices is a specialisation of an implementation approach for service-oriented architectures (SOA) used to build flexible, independently deployable software systems. Services in a microservice architecture are communicating with each other over the network in order to fulfill customer's requirements. Studies and discussions are needed for improvement of microservices system including: time efficiency, cost efficiency, system maintainability, rationality of system architecture and etc.

In this project we have three goals, First is to build a microservices system which utilized RESTful APIs to communicate with each other, Second is to build up a monitor service which can collect all the communication informations within the microservice system. Finally we hope to translate and display the captured data for future study.

2. Motivation

In order to build up a microservices tracking system we need to build up a microservices system first. Spring is the one of the most ad-hoc and best framework to build up distributed systems like microservices. It provides tools for developers to quickly build some of the common patterns in distributed systems. And since our demo system need to have multiple independent microservices and contains scenarios that involves at least two services' RESTful communication, so we will need supporting tools for building up communication link. Spring Cloud can provide good out of box experience for typical use cases and extensibility mechanism to cover others. So it would be easier for us to build our own customized microservice using Spring, Spring Boot and Spring Cloud

For the purpose of tracking these API calls between services, we will need two important configurations: we need to configure each of the service into trace emitter, and we need to set up a trace collector to capture and store those communication traces. Spring Sleuth and Zipkin are the best chose for us. As both of these libraries are open source code, study curve will be a little bit steep for us. However, this also provide us a chance to dig into the code and utilize the fundamental features of Spring Sleuth and Zipkin.

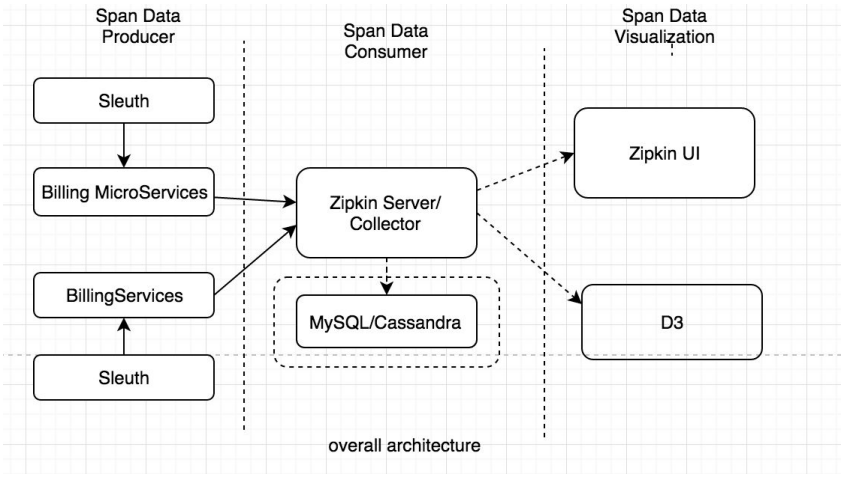
For displaying and future study of the relationships between services, get and translate trace data into JSON format is necessary. This requires us to dig into the most detailed and basic working principle of the Sleuth and zipkin and analyze the composition of every trace. Based on this we can obtain, convert and display the traces we are interested in and start future works.

3. Related work

Related study and work focus on the following topics:
Maven project building, running and debugging;
Maven dependencies management;
Google Dapper;
Spring-Boot;
Spring-Cloud;
Spring-Cloud-Sleuth;
Spring-Cloud-Stream;
RabbitMq;
Zipkin.

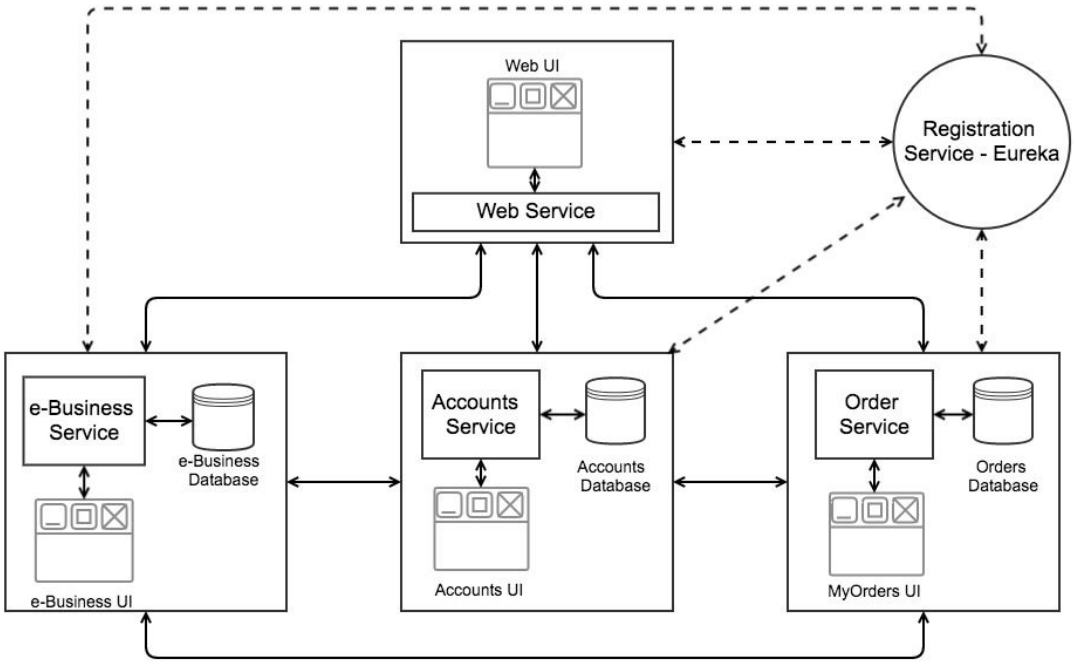
4. System design

In this section we will introduce the design of our system which contains three parts: Firstly we will introduce the design of our Billing microservices system. Secondly, we will describe the structure of Distributed tracking facilities implemented in our system. Finally we will provide the designs of our Traces' collection and display services. Overall architecture of the our project system is shown in the following figure.



4.1 Billing Microservices system design

We built up our own demo micro-services system as a Billing Services system. This Billing Services System contains five main parts including: Registration service, Accounts service, Web service, e-Business service and Order service. Overview system design is shown in the following Figure.



The Billing microservices system serves as a billing service platform which provides services like paying orders, check user accounts and e-Business accounts management and etc. As shown in above figure, the Billing micro-services system contains five parts. Web services, e-Business service, accounts service and order service are functional services which are responsible for part of the system's functionality. Each micro-service contains its own front-end components such as HTML pages and controllers and also back-end components such as services, configurations and etc. In each service, front end talk to the back end through REST APIs. Between services, they also talk to each other through REST APIs. The implementation details will be provided in section 5. There is also a registration service - Eureka, which is implemented as a services registry and help users and administrators to found and manage services.

Detailed functionalities and connections between services are introduced in the following sections.

4.1.1 Registration Service - Eureka

In our system, Eureka serves as a phone book for all the microservices. Each service registers itself with the service registry and tells the registry where it lives (host port, node name and etc.) and perhaps other service-specific metadata which can be used by other services to make informed decisions about it. Clients can ask questions about the service topology and service capabilities. Furthermore, developer may use technology that has some notion of a cluster and that information is ideally stored in a service registry.

In our Billing Micro-services system, all the other functional services are registered in the Eureka registry with detailed and necessary informations.

4.1.2 Web Service

Web service serves as API gateway in our system. It is connected with other functional services such as Accounts service, e-Business service and Order service. It provides a centralized UI for users and developers to test and demo the functionalities as well as the APIs between each service. Web service can talk to e-Business service, Accounts service and Order service. It can get related informations and display through Web service's UI.

4.1.3 Accounts Service

Accounts service stores users' detailed account information, such as account ID, user name, e-Business account ID and etc. This user account is the account in the platform and serves as an index of each user's detailed information. For example, from accounts server, we can access one's e-business account and check the balance information and etc. Accounts service contains its own front-end UI and backend database. The Accounts service UI is used for demo and tests of its functionalities such as checking a user's order history, account information and etc.. All this data are stored in the Accounts service database.

4.1.4 e-Business Service

E-Business service is responsible for users' economic activities such as check balance, pay the user's order with certain bank account and etc. The e-Business service serves the user's bank account and contains the informations such as bank account number, user name, account balance and etc. Similar as Accounts service, e-Business service also contains its own UI and backend database. In the front-end, e-Business service provide UI for users to: Update certain user e-business account, check a certain e-business account, check all users' e-business accounts and etc. These informations are all stored in the backend database.

4.1.5 Order Service

Order service works when a user choose to pay an order. Order service connects with e-business accounts. When an certains order is selected and user choose to pay the order, the order object contains the item ID and price will be generated and sent to the e-Business account. After that, the user will be charged from his/her bank account and the balance will be decreased correspondingly. Order service also contains its own UI for testing for its functionalities and database for storing order history information.

4.2 Distributed Tracking

Spring Cloud Sleuth is a tracing solution for Spring Cloud. Its concepts are borrowed from Google Dapper[1], Zipkin and HTrace. Sleuth should be invisible to most users, which means that the interactions between the services will be automatically taken care of most of the time. And the Sleuth data can be used in the forms of logs, or collected by a remote service[1]. In our system, we use Sleuth to instrument the services and send it to the remote collector.

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data. Zipkin's design is based on the Dapper. Zipkin is widely used in the Industry. Twitter also provides a presentation about Zipkin's usage in Twitter[] by Jeff Smick, the link provided for your reference. We can easily see Zipkin as the combination of of collectors, consumers and can present the Sleuth Data in the UI.

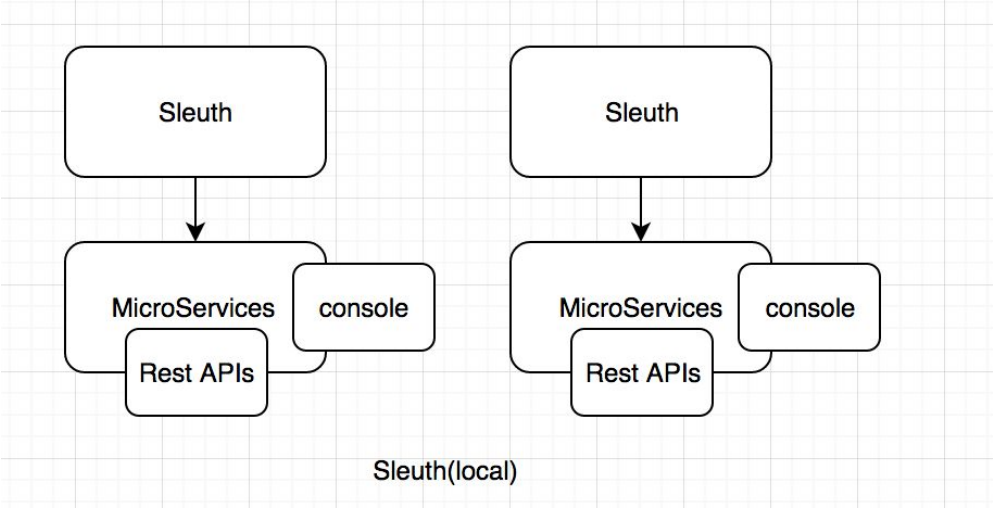
RabbitMQ is an open source message broker software that implements the Advanced Message Queuing Protocol(AMQP).The RabbitMQ server is written in the Erland. and its client libraries to interface with the broker are available for all major programming languages.[]

In our project, we have tried at least four ways to play with Sleuth and Zipkin. Hope you can enjoy the innovativity and convenience of the technologies.

1) Sleuth+exposed by UI
we produced the Span data for the web services and then log out the data locally- in the console. At the same time, we expose the Span data by the REST APIs.
2)Sleuth Stream+RabbitMQ+Zipkin
We use Sleuth Stream as the Span data producer, the send the Span data to RabbitMQ, which will finally consumed by Zipkin.
3) Sleuth Stream + RabbitMQ + Custom Consumer
We use Sleuth Stream as the Span data producer, and then send the Span data to RabbitMQ, which finally would be consumed by the consumer program we write.
4) Sleuth+Zipkin
Sleuth Produce the Span data, and then directly send the data to the Zipkin. Zipkin will collect the show the data in the UI.

4.2.1 Sleuth+exposed by REST APIs

we produced the Span data for the web services and then log out the data locally in the console. At the same time, we expose the Span data by the Rest APIs. The easiest way to get a sense of Zipkin data. However, it's not an easy way to collect the Span data for further use. So we might not talk too much about this in this report.

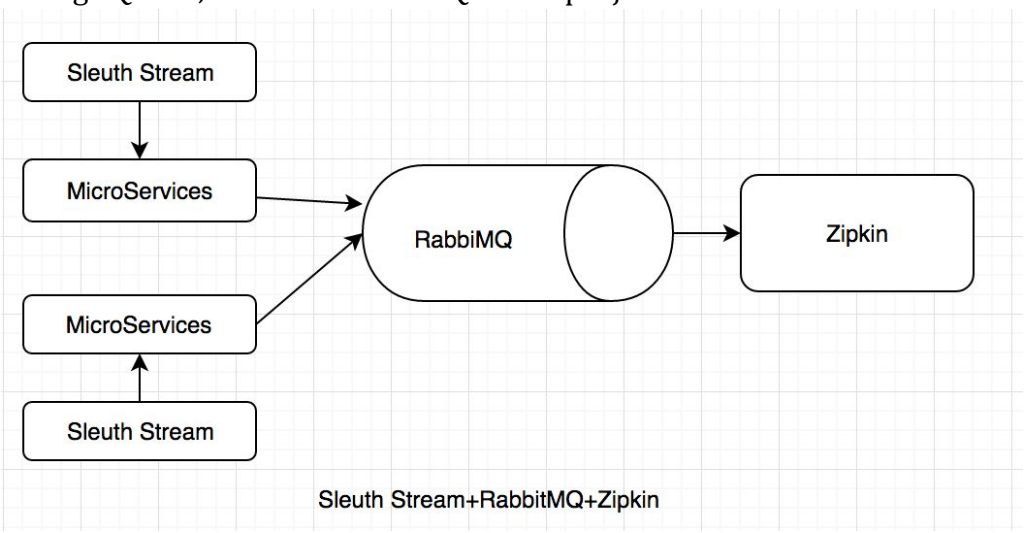


The above figure shows that two microservices exposing Span data both in the local console and backend APIs.

4.2.2 Sleuth Stream + RabbitMQ + Zipkin

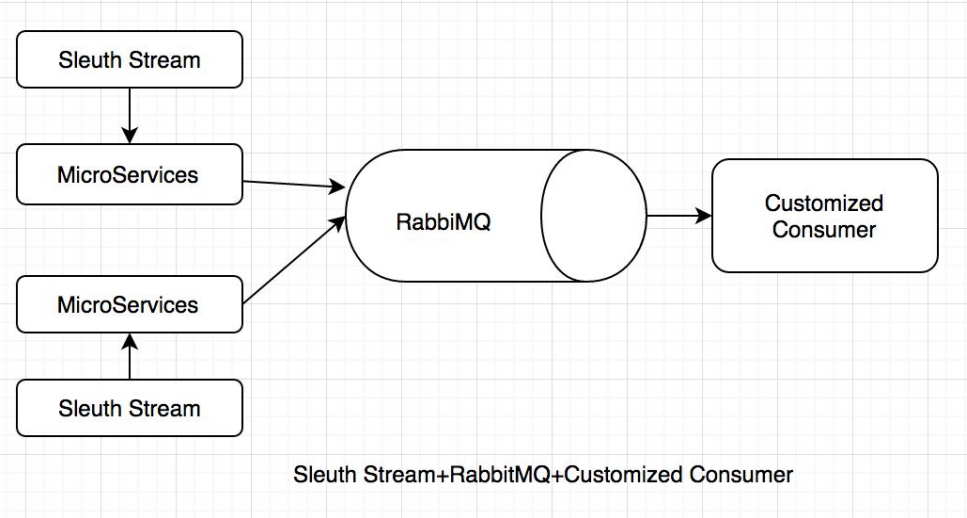
We use Sleuth Stream as the Span data producer, the send the Span data to RabbitMQ, which will finally consumed by Zipkin.

Sleuth Stream is built upon Spring-Cloud-Sleuth and Spring-Cloud-Stream, so it can subscribe the channels in the Message Queue and produce the Span data and send to the Message Queue, which is RabbitMQ in our project.

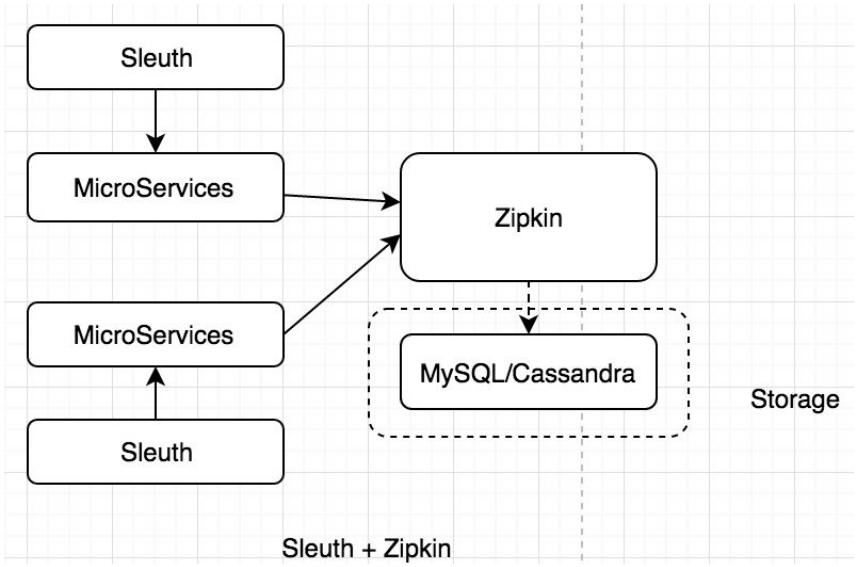


4.2.3 Sleuth Stream + RabbitMQ + Customized Customer

Similar to the previous part, we use Sleuth Stream as the Span data producer, and then send the Span data to RabbitMQ; However, at the consumer part, we use our customized consumer to collect the Span data. The reason behind this is that Zipkin is a quite complex system, if we want to tail our system to our specific needs, we can write the consumer programs ourselves.



4.2.3 Sleuth + Zipkin



Sleuth Produce the Span data, and then directly send the data to the Zipkin. Zipkin will collect the data, and show the data in the zipkin UI. Obviously, it's very handy to use this framework to implement the distributed tracing. While some configurations should be done to leverage the Zipkin usage, especially for the storage part. Zipkin supports multiple databases, like Cassandra, MySQL, ElasticSearch etc. But the default mode is the in-memory database. We will talk more about the Zipkin storage in the following implementation chapter.

4.3 DB collection and Display

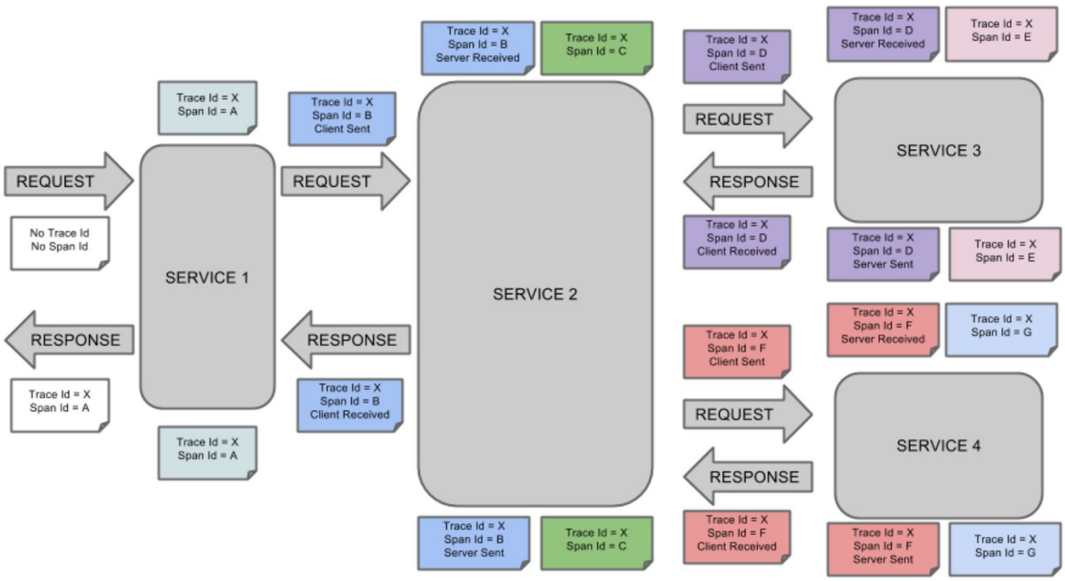
By using Sleuth and Zipkin, we were able to track and collect the communications information in the network. These information are wrapped and emitted by each service in the form of "Traces" and "Spans". Thanks to Sleuth and zipkin, we are also able to store these data into local database such as MySQL. However, we still need to collect and translate these "Traces" and "Spans" to certain customized data type such JSON if we hope to select, compare, analyze or display these tracks. The following sections will provide our design about how to collect and process these data and how to display these traces through front end framework - D3.

4.3.1 DB collection and processing

4.3.1.1 Data Structure used in Sleuth

1. Traces:
A set of spans forming a tree-like structure. For example, if you are running a distributed big-data store, a trace might be formed by a put request.

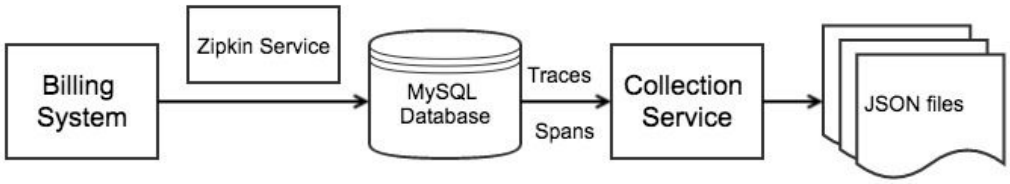
2. Spans:
In Spring Cloud Sleuth, a "Span" is a basic unit of work, Span's are identified by a unique 64-bit ID for the span and another 64-bit ID for the trace the span is a part of. Spans also have other data, such as descriptions, timestamped events, key-value annotations (tags), the ID of the span that caused them, and process ID's (normally IP address). Spans are started and stopped, and they keep track of their timing information. The Traces and Spans of a sample Request in a micro-services system is illustrated as below:



3. Annotation:
Annotations are used to record existence of an event in time. It is the most important part of a Span. There are two kinds of annotations: Annotation and Binary-Annotations. Annotations are used to record the existence of an event in time manner. Binary-Annotations are used to record an event's other informations such as the host and port of the service, method of the request, Path and URL of the request and etc.

4.3.1.2 Design of the Collection Service

In order to collect the traces and spans data from database and translate them into JSON file for future study, we build up another microservice which can be used to read spans and annotations from database "zipkin" and sort and process them into JSON file. This "Collection Service" has its own UI and have the functionalities such as read certain trace from database, delete data from database and etc. The service can also write the JSON data out to JSON files. Its implementation details are described in section 5.

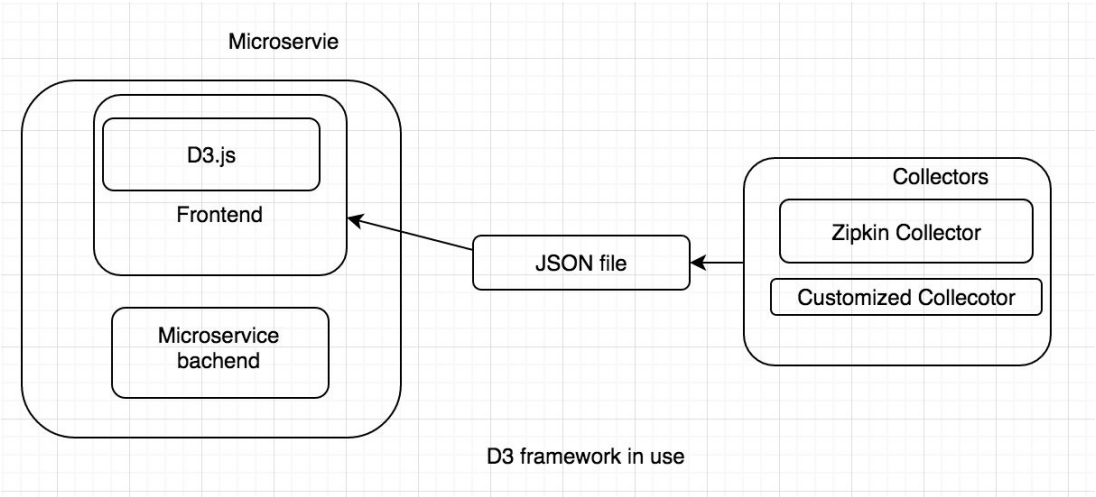


4.3.2 D3 Display

D3 (Data-Driven Documents or D3.js) is a JavaScript library for visualizing data using web standards. D3 helps you bring data to life using SVG, Canvas and HTML. D3 combines powerful visualization and interaction techniques with a data-driven approach to DOM manipulation, giving you the full capabilities of modern browsers and the freedom to design the right visual interface for your data[]. The benefits of using D3 can create a UI components that can interact with users.

D3 can consume data from various form, for example, csv, tsv, or json file. The APIs of D3 are provided online. Also, there are a lot of examples which we can learn from online.

We use D3 in our frontend to display the Span Data stored in the json file. This file could be from Zipkin collector(A component of Zipkin) or the collector we built as another microservice.

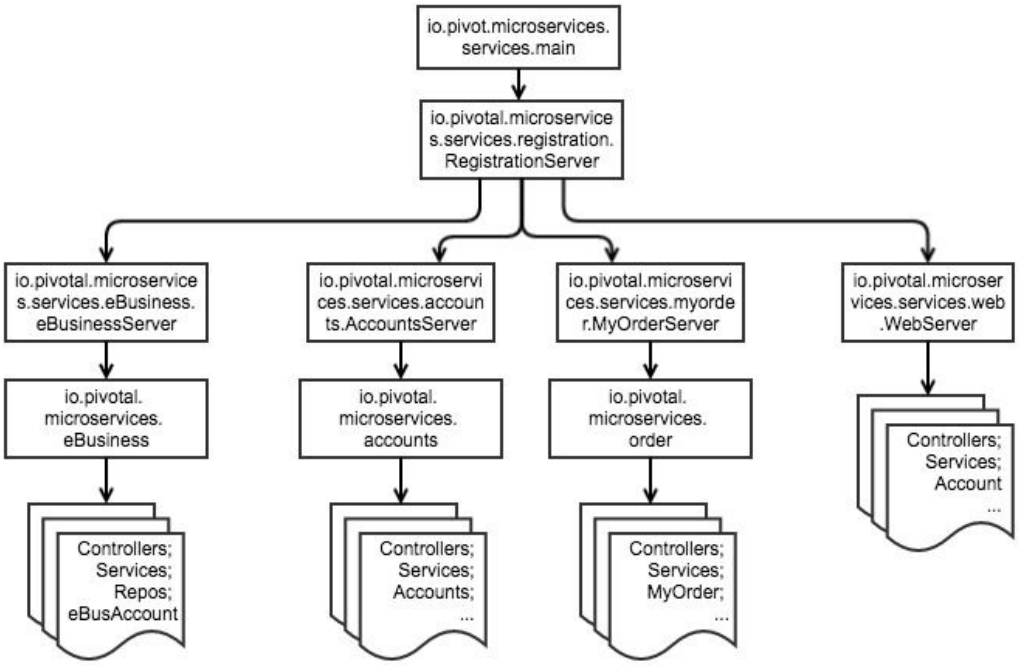


5. System Implementation

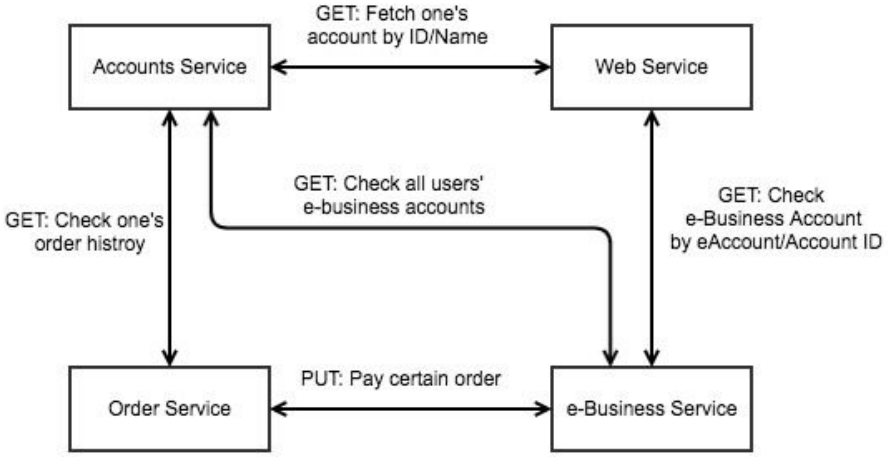
5.1 Billing microservices system implementation

This Billing microservices system is implemented in one project folder. Each microservice is a Spring Boot application and contains its own configuration .yml file. In this file we configure the service' name, running port, Eureka configuration, Zipkin configuration, Sleuth configuration and etc.

The whole project works as a maven project and requires a "mvn clean install" to install dependencies, compile and produce .jar file. The project folder structure of Billing microservice system is shown as below:



Functional REST APIs between services are described in the following Figure:

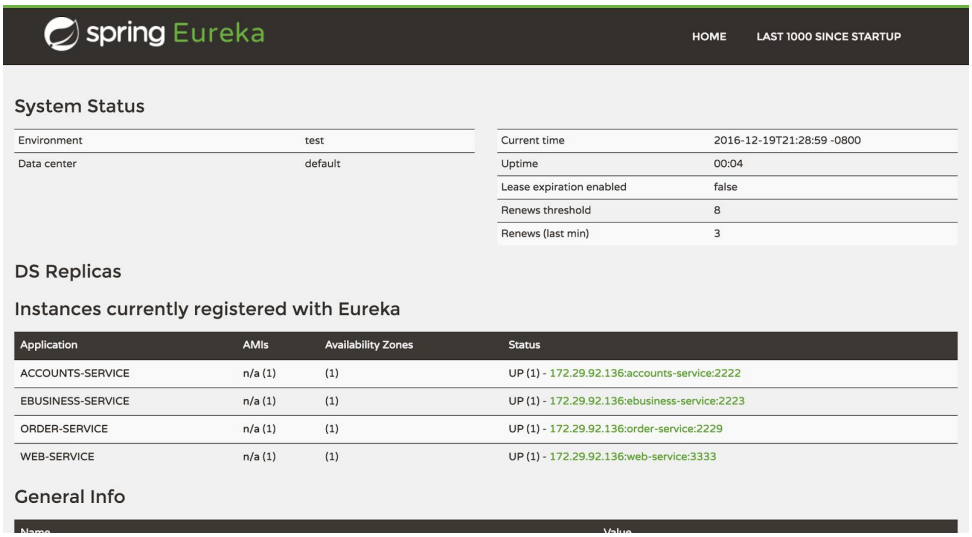


5.1.1 Services Registration - Eureka

Implement Eureka registry is simple: By claiming "@EnableEurekaServer" and "SpringBootApplication" and run the main function, the Eureka server will get start running.

```java
@EnableBinding(SleuthSink.class)
public class SampleSleuthApplicationConsumer {

    public static final String CLIENT_NAME = "testAppConsumer";
    private static final Log logger = LogFactory.getLog(SampleSleuthApplicationConsumer.class);


    @StreamListener(SleuthSink.INPUT)
    public void logSink(Spans input) {
        logger.info("In consumer: the Spans are: " + input.getSpans().toString());
    }
}
```

After all services' get start running, we can find the registrations in Eureka's UI.



5.1.2 Accounts Service:

Back-end Implementation:

Accounts service's major function files are in this folder:

- Account.java is the account object maintained in the system which contains all the informations about a user's account in the system including: userID, user name and etc. As it is a persistent Entity supported Spring JPA, this object can be directly connected to the service's database through "AccountRepository" which is serving as an JPA API between Account service and its database.

```java
@Entity
@Table(name = "T_ACCOUNT")
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    public static Long nextId = 0L;

    @Id
    protected Long id;

    protected String idnumber;

    @Column(name = "name")
    protected String owner;

    protected BigDecimal balance;

    protected static Long getNextId() {
        synchronized (nextId) {
            return nextId++;
        }
    }

    protected Account() {
        balance = BigDecimal.ZERO;
    }

    public Account(String number, String owner) {
        id = getNextId();
        this.idnumber = number;
        this.owner = owner;
        balance = BigDecimal.ZERO;
    }
```

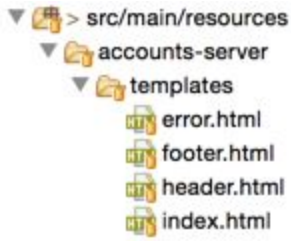
- AccountRepository is the API for database which extends Spring Cloud Repository<Account, Long>. This API interface provides us a simple and fast way to access database.

```java
public interface AccountRepository extends Repository<Account, Long> {
    /**
     * Find an account with the specified account number.
     *
     * @param accountNumber
     * @return The account if found, null otherwise.
     */
    public Account findByIdnumber(String accountNumber);

    public ArrayList<Account> findAll();
    /**
     * Find accounts whose owner name contains the specified string
     *
     * @param partialName
     *            Any alphabetic string.
     * @return The list of matching accounts - always non-null, but may be
     *         empty.
     */
    public List<Account> findByOwnerContainingIgnoreCase(String partialName);

    /**
     * Fetch the number of accounts known to the system.
     *
     * @return The number of accounts.
     */
    @Query("SELECT count(*) from Account")
    public int countAccounts();
}
```

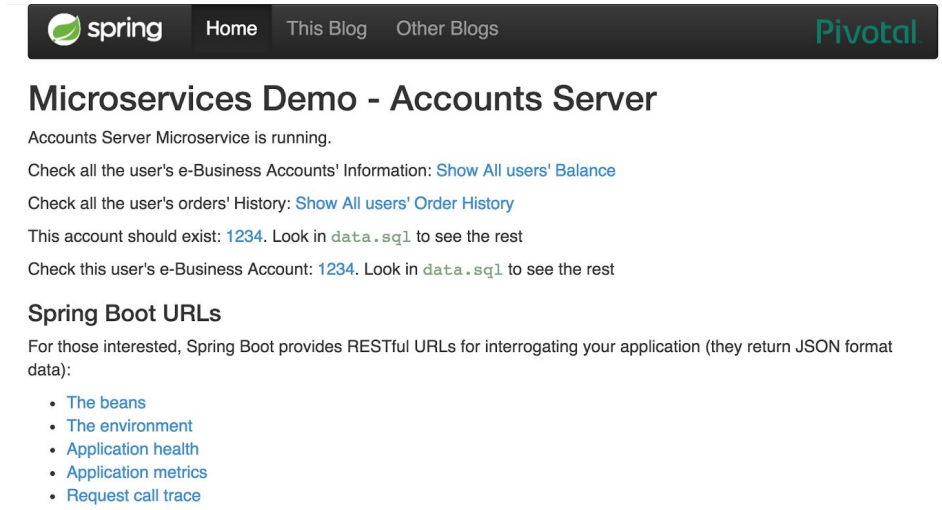
- AccountsController is the API controller for Accounts service. Please refer to the code for the usages of these APIs.

Front-end:

Front end components are in the src/main/resources/templates folder.



UI of the Accounts service are as follows:



5.1.3 Order Service

Back-end Implementation:

Order service also contains:

- MyOrder: MyOrder is an Object defined in order service and maintained as an Spring Cloud Entity which is directly linked to the order service database. It contains all the properties as order information such as ordeID, product name, price and etc.

- MyOrderRepository: MyOrder Repository provides APIs for the myorder service to access the database and connect it with MyOrder Entity. Sample queries are shown below:

```
@Query("SELECT count(*) from MyOrder")
public int countOrders();

public MyOrder findByid(Long orderId);

public List<MyOrder> findByUsrId(String userId);

public List<MyOrder> findAll();

@Modifying
@Query("update MyOrder myor set myor.usrId = ?1 where myor.usrId = '100000000'")
public int setFixedUserIdFor(String usrId);

@Modifying
@Transactional
//should use the origin name in the class definition rather than the column name
@Query("update MyOrder myor set myor.payStatus = 'Paid' WHERE myor.id = ?1")
public void updateStatusToPaid(Long orderId);


@Modifying
@Transactional
@Query("UPDATE MyOrder myor set myor.payStatus = 'Unsucc' WHERE myor.id = ?1")
public void updateStatusToUnsuccessful(Long orderId);
```

- MyOrderController: MyOrderController is a Front-end API controller. It is used to handle the REST APIs calls from myorder service's UI. This controller also works as a router which connects the front end requirements to the MyOrderService's functions.
- MyOrderService:Back-end service function file. This file handles all the the RESTful APIs communications between order service and other services.

Front-end:

Below is the orders selection and payment operation. From this table user can choose any of the item and pay. When "Pay" button is clicked, A PUT request will be sent to e-Business service and if the user's balance is enough, the order will be paid successfully, otherwise a warning saying balance not enough will be shown.

All the orders

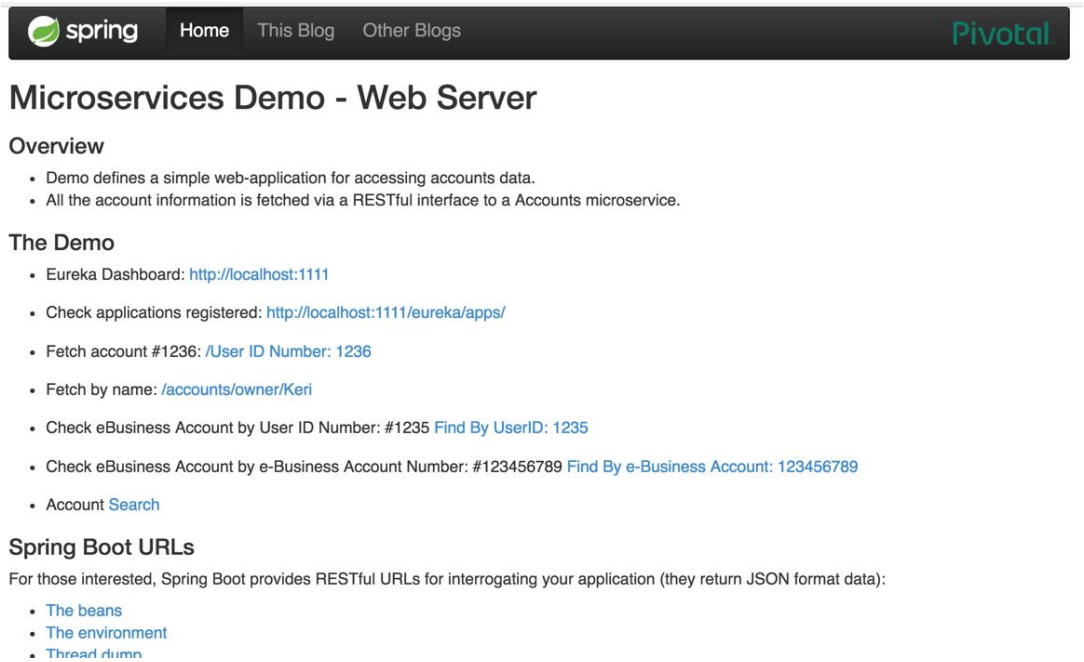
Order Id	User Id	Product Name	Price	Pay Status	
0	1234	Nike Shoes	100.00	Unpaid	Pay
1	1234	Adidas Clothes	101.00	Unpaid	Pay
2	1234	Tooth	1.00	Unpaid	Pay
3	1235	Free item	0.00	Unpaid	Pay
4	1235	Keyboard	200.00	Unpaid	Pay
5	1235	Mouth	20.00	Unpaid	Pay
6	1236	Phone	500.00	Unpaid	Pay
7	1237	Bike 1	200.00	Unpaid	Pay
8	1238	Bike 2	300.00	Unpaid	Pay
9	1239	Bike 3	400.00	Unpaid	Pay
10	1278	Bike 3	400.00	Unpaid	Pay

5.1.4 Web Service

Web service doesn't contains database. Its front-end controllers and other files are shown in the following Figure:

The WebAccountController and WebEBusinessAccountsController are controllers handling front-end account and e-business account requirements respectively. The services are handling the REST API calls to the corresponding services such accounts service or e-Business account service.

Front-end Implementation of the Web service is shown in the following figure:



For example, when we choose "Fetch account #1236" and click on the Link, a GET request will be sent to account service and Web service will get the required Account Entity and display it as the following Figure.

5.2 Distributed Tracing

In this part, we will illustrate how to implement different Distributed Tracing strategies mentioned in Chapter 4.2.

We have tried four different ways use Span data. Worth mentioning that some documents from the community is not clear enough. For simplicity of learning, we will explain how to implement the first three ways by using the sample code from community[], especially for those who are new to Spring and Spring-boot.

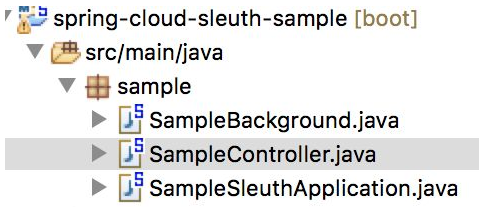
5.2.1 Sleuth+exposed by UI

we produced the Span data for the web services and then log out the data locally- in the console. At the same time, we expose the Span data by the REST APIs. Dependency management by mvn:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

the core part of the pom file.

This project is a typical Spring MVC framework.



the structure of the demo code

```
@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass = true)
@EnableAsync
public class SampleSleuthApplication {

    public static final String CLIENT_NAME = "testApp";

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean
    public SampleController sampleController() {
        return new SampleController();
    }

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

}
```

Spring Framework actually is a Factory of Beans, which means it creates and configures Beans to implement certain functions by Code Injection. Spring Cloud is built upon Spring Boot. Basically every microservice is a Spring Boot application. The above code snippet is main function unit of this Spring application. This snippet creates some Beans which are @Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports most of the attributes offered by <bean/>.

Controller

```
 */
@RestController
public class SampleController
        implements ApplicationListener<EmbeddedServletContainerInitializedEvent
    private static final Log log = LogFactory.getLog(SampleController.class);
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private Tracer tracer;
    @Autowired
    private SpanAccessor accessor;
    @Autowired
    private SampleBackground controller;

    private Random random = new Random();
    private int port;

    @RequestMapping("/")
    public String hi() throws InterruptedException {
        Thread.sleep(this.random.nextInt(1000));

        String s = this.restTemplate
                .getForObject("http://localhost:" + this.port + "/hi2", String.
        return "hi/" + s;
    }
```

It uses @RestContoller annotation to create a controller, within which implements RestAPIs annotated by RequesteMapping. Note that: the Tracer and SpanAccessor are declared here and used here to add annotations to the Span Data.

```
@RequestMapping("/call")
public Callable<String> call() {
    return new Callable<String>() {
        @Override
        public String call() throws Exception {
            int millis = SampleController.this.random.nextInt(1000);
            Thread.sleep(millis);
            SampleController.this.tracer.addTag("callable-sleep-millis",
                    String.valueOf(millis));
            Span currentSpan = SampleController.this.accessor.getCurrentSpan();
            return "async hi: " + currentSpan;
        }
    };
}
```

in the above snippet, it defines an web endpont “/call”, and assocaites it with the “call” method. In the call method, it add binary annotations to the Span Data. Annotations of the Span Data is key-value pairs. Here, the key is “callable-sleep-millis”, value is the random number of the wait times. And this API will expose the API by return currentSpan as result.

SampleController.java

```
@Component
public class SampleBackground {

    @Autowired
    private Tracer tracer;
    @Autowired
    private Random random;

    @Async
    public void background() throws InterruptedException {
        int millis = this.random.nextInt(1000);
        Thread.sleep(millis);
        this.tracer.addTag("background-sleep-millis", String.valueOf(millis));
    }

}
```

This controller defines a async background component to add binary annotations.

5.2.2 Sleuth Stream+RabbitMQ+Zipkin

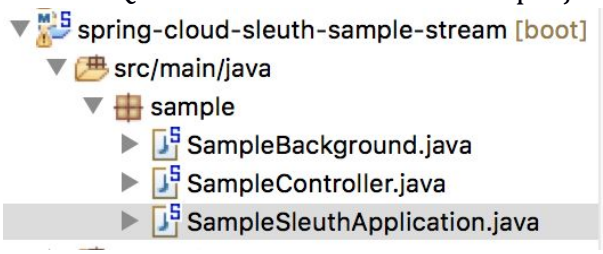
In this part, we use Sleuth Stream as the Span data producer, the send the Span data to RabbitMQ, which will finally consumed by Zipkin.

The example is sample-stream, which will send Span Data to MessageQueue. The pom file is where the differs from the Sleuth.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-stream</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.integration</groupId>
        <artifactId>spring-integration-jmx</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
```

As you can see, the Spring-cloud-sleuth-stream replace the spring-cloud-sleuth-core in the previous project, and it import a package called

spring-cloud-stream-binder-rabbit, which specifies the AMCQ this project uses is RabbitMQ. And the main code for this project is identical from the previous project.



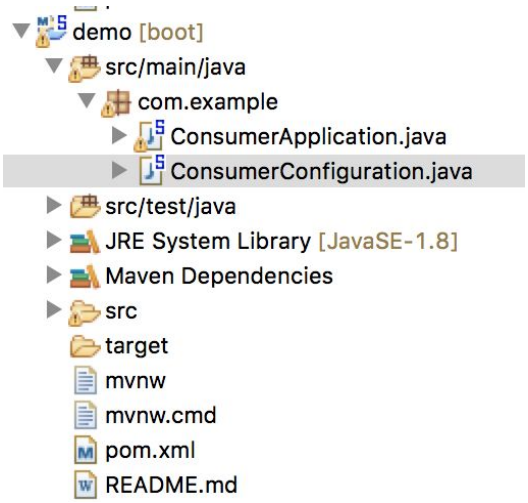
Note that: to successfully consume the data successfully, we have to a running RabbitMQ and Zipkin. RabbitMQ and Zipkin should be downloaded online and installed and run successfully!!! Since we used MacOS, we used homebrew to install RabbitMQ and the Zipkin could be downloaded from zipkin official website[]. Some details would be provided in the tutorial part.

5.2.3 Sleuth Stream + RabbitMQ + Custom Consumer

The Sleuth Stream will send Span Data as Stream Messages. Hence, in this project, We use the spring-cloud-sleuth-sample-stream as our Span Data producer, and then send the Span data to RabbitMQ, which finally would be consumed by the consumer program we write.

Note that: the tutorial from the Sleuth tutorial[] is outdated. We manage to implement the customized consumer by reading the Sleuth source code and communicating with the contributors of Spring Cloud Sleuth. If you want to implement your customized consumer, you might need spend a lot time. This is also the reason we don't use this way to document system, which means you have to build a mini-zipkin almost from scratch. Anyway, we illustrate how to to a customized consumer here, hoping helping you understand the technologies behind Zipkin better.

The Span Data we use the same project of Spring-cloud-sleuth-sample-stream as the previous part. Still, we need a running RabbitMQ. And the running consumer project might look like this:



pom.xml:

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

    <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-sta
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
    </dependency>

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

We use the @StreamListener annotation to register our consumer and automatically convert the message to Span data.

```
@EnableBinding(SleuthSink.class)
@Import(ConsumerConfiguration.class)
@SpringBootApplication(exclude = SleuthStreamAutoConfiguration.class)
public class ConsumerApplication {

    private static final Log log = LogFactory.getLog(ConsumerApplication.class);
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

    @StreamListener(SleuthSink.INPUT)
    public void sink(Spans input) throws Exception {
        log.info("In consumer: the Spans are: " + input.getSpans().toString());
    }

}
```

We import a configuration file to customize the Span Reporter:

```
@Configuration
public class ConsumerConfiguration {
    @Bean(name = StreamSpanReporter.POLLER)
    PollerMetadata customPoller() {
        PollerMetadata poller = new PollerMetadata();
        poller.setMaxMessagesPerPoll(500);
        poller.setTrigger(new PeriodicTrigger(5000L));
        return poller;
    }
}
```

5.2.4 Sleuth+Zipkin

Sleuth Produce the Span data, and then directly send the data to the Zipkin. Zipkin will collect the show the data in the UI. It's a mature solution to implement the distributed tracing. Hence, we choose this solution to instrument mini-billing system microservices and practise with Spring Cloud Sleuth and Spring Cloud Zipkin.

To utilize Sleuth and zipkin in our system, part of pom.xml file is shown as below:

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

To configure the service into a spans emitter and make sure the data are stored into local database, the .yml file of each service should also be modified. Take Account service as an example:

```
spring:
  application:
    name: accounts-service  # Service registers under this name
  zipkin:
    baseUrl: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
  freemarker:
    enabled: false          # Ignore Eureka dashboard FreeMarker templates
  thymeleaf:
    cache: false            # Allow Thymeleaf templates to be reloaded at runtime
    prefix: classpath:/accounts-server/templates/    # Trailing / mandatory

sample:
  zipkin:
    # When enabled=false, traces log to the console. Comment to send to zipkin
    # enabled: false
```

By doing this, the configured service is turned into a spans emitter and allows the zipkin to collect every span of its RESTful communications in the network.

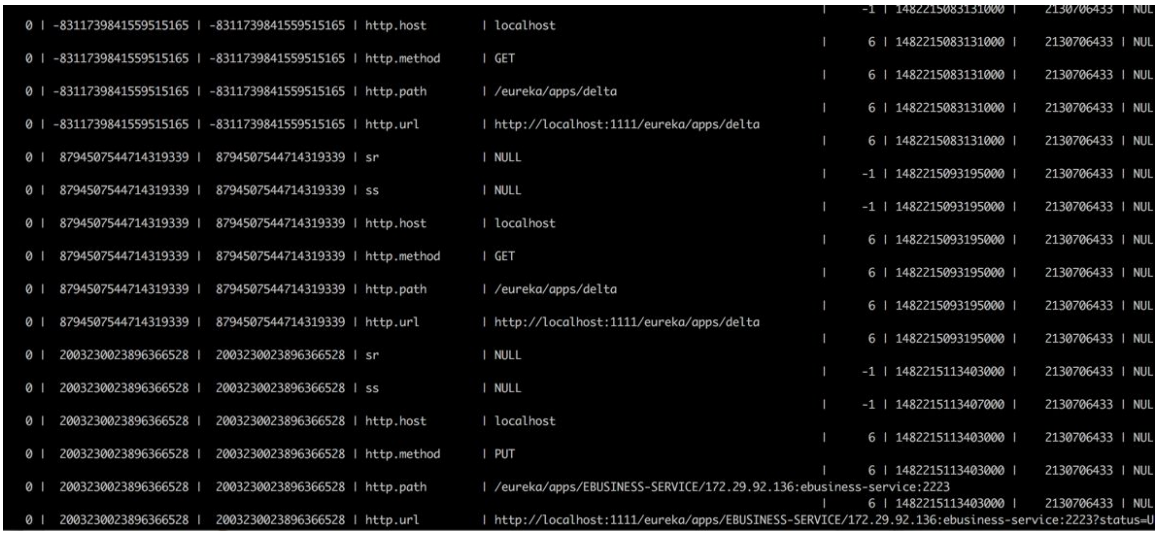
5.3 DB collection and Display

To collect the trace data and translate them into JSON objects or files, we build a DB collection and processing service named “Collector-Service”. Implementation of the Collector Service will be introduced in the following section.

5.3.1DB collection and processing

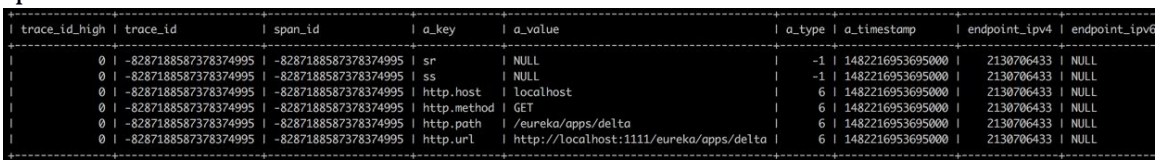
After services and zipkin collector are configured correctly and get start running, spans and annotations among the services will be collected and stored in local MySQL database as shown in the following figures:

Annotations:



0	-8311739841559515165	-8311739841559515165	http.host	localhost	-1	1482215083131000	2130706433	NUL
0	-8311739841559515165	-8311739841559515165	http.method	GET	6	1482215083131000	2130706433	NUL
0	-8311739841559515165	-8311739841559515165	http.path	/eureka/apps/delta	6	1482215083131000	2130706433	NUL
0	-8311739841559515165	-8311739841559515165	http.url	http://localhost:1111/eureka/apps/delta	6	1482215083131000	2130706433	NUL
0	8794507544714319339	8794507544714319339	sr	NULL	6	1482215083131000	2130706433	NUL
0	8794507544714319339	8794507544714319339	ss	NULL	-1	1482215093195000	2130706433	NUL
0	8794507544714319339	8794507544714319339	http.host	localhost	-1	1482215093195000	2130706433	NUL
0	8794507544714319339	8794507544714319339	http.method	GET	6	1482215093195000	2130706433	NUL
0	8794507544714319339	8794507544714319339	http.path	/eureka/apps/delta	6	1482215093195000	2130706433	NUL
0	8794507544714319339	8794507544714319339	http.url	http://localhost:1111/eureka/apps/delta	6	1482215093195000	2130706433	NUL
0	2003230023896366528	2003230023896366528	sr	NULL	6	1482215093195000	2130706433	NUL
0	2003230023896366528	2003230023896366528	ss	NULL	-1	1482215113403000	2130706433	NUL
0	2003230023896366528	2003230023896366528	http.host	localhost	-1	1482215113407000	2130706433	NUL
0	2003230023896366528	2003230023896366528	http.method	PUT	6	1482215113403000	2130706433	NUL
0	2003230023896366528	2003230023896366528	http.path	/eureka/apps/EBUSINESS-SERVICE/172.29.92.136:ebusiness-service:2223	6	1482215113403000	2130706433	NUL
0	2003230023896366528	2003230023896366528	http.url	http://localhost:1111/eureka/apps/EBUSINESS-SERVICE/172.29.92.136:ebusiness-service:2223?status=U	6	1482215113403000	2130706433	NUL

Spans:



trace_id_high	trace_id	span_id	a_key	a_value	a_type	a_timestamp	endpoint_ipv4	endpoint_ipv6
0	-8287188587378374995	-8287188587378374995	sr	NULL	-1	1482216953695000	2130706433	NULL
0	-8287188587378374995	-8287188587378374995	ss	NULL	-1	1482216953695000	2130706433	NULL
0	-8287188587378374995	-8287188587378374995	http.host	localhost	6	1482216953695000	2130706433	NULL
0	-8287188587378374995	-8287188587378374995	http.method	GET	6	1482216953695000	2130706433	NULL
0	-8287188587378374995	-8287188587378374995	http.path	/eureka/apps/delta	6	1482216953695000	2130706433	NULL
0	-8287188587378374995	-8287188587378374995	http.url	http://localhost:1111/eureka/apps/delta	6	1482216953695000	2130706433	NULL

As the spans and annotations are stored in a random order, we cannot use these data to find out the time traces of each REST events. Therefore, we need to categorize these spans and annotations into different traces and translate them into JSON object. First we need to understand the meaning of basic units in the span - Annotations. Annotations contains two form: Annotations and Binary-Annotations. A span can contains only one of them or both of them. Explanations are as follows:

Annotations:

- cs - Client Sent - The client has made a request. This annotation depicts the start of the span.
- sr - Server Received - The server side got the request and will start processing it. If one subtracts the cs timestamp from this timestamp one will receive the network latency.
- ss - Server Sent - Annotated upon completion of request processing (when the response got sent back to the client). If one subtracts the sr timestamp from this timestamp one will receive the time needed by the server side to process the request.
- cr - Client Received - Signifies the end of the span. The client has successfully received the response from the server side. If one subtracts the cs timestamp from this timestamp one will receive the whole time needed by the client to receive the response from the server.

A typical Annotation is as follows:

Date Time	Relative Time	Annotation	Address
12/19/2016, 9:49:09 PM	120.000ms	Client Send	172.29.92.136:2222 (accounts-service)
12/19/2016, 9:49:13 PM	3.477s	Server Receive	172.29.92.136:2223 (ebusiness-service)
12/19/2016, 9:49:14 PM	4.840s	Server Send	172.29.92.136:2223 (ebusiness-service)
12/19/2016, 9:49:14 PM	5.026s	Client Receive	172.29.92.136:2222 (accounts-service)

Binary-Annotations:

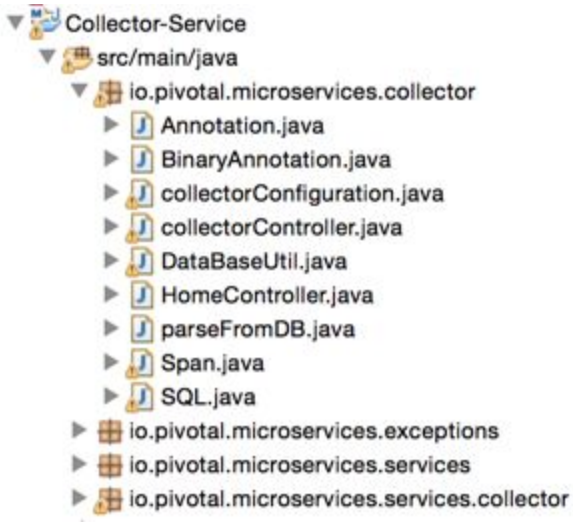
- Key: name of the tag, including the current REST calls': host, method, path, component and etc.
- Value: Values of the above keys.

A typical Binary-Annotation is as follows:

Key	Value
http.host	172.29.92.136
http.method	GET
http.path	/eBusinessAccount/findAll
http.url	http://172.29.92.136:2223/eBusinessAccount/findAll
Local Component	unknown
Local Address	172.29.92.136:2223 (ebusiness-service)

Based on the Tree-Like structure of a trace, we were able to interpret the spans' relationships by its current span_id and parent span_id. According to the inheritance relationships, we were able to build up each trace tree and complete the detailed information of each spans in the trace.

Major files of the Collector-Service are shown in the following figure:

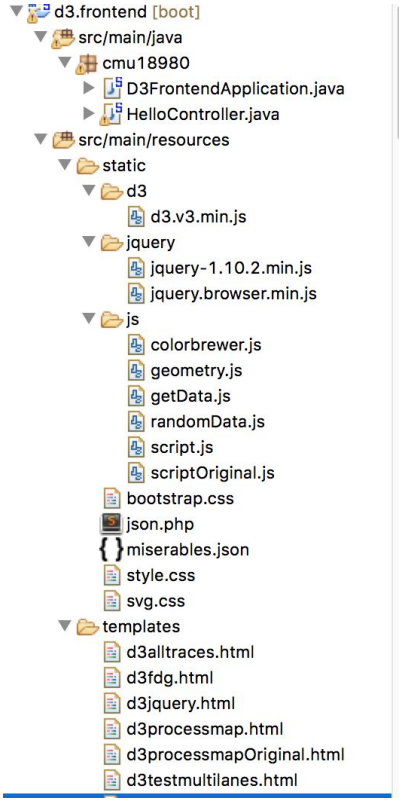


One translated trace as a JSON file is shown in the following Figure:

```
[{"traceId":"6681da9173280aa7","id":"6681da9173280aa7","name":"http:/accounts/showall","timestamp":1482212949672000,"duration":504500
0,"annotations":[{"timestamp":1482212949672000,"value":"sr","endpoint":{"serviceName":"accounts-
service","ipv4":"172.29.92.136","port":2222}},{"timestamp":1482212954717000,"value":"ss","endpoint":{"serviceName":"accounts-
service","ipv4":"172.29.92.136","port":2222}}],"binaryAnnotations":[{"key":"http.host","value":"172.29.92.136","endpoint":
{"serviceName":"accounts-service","ipv4":"172.29.92.136","port":2222}},{"key":"http.method","value":"GET","endpoint":
{"serviceName":"accounts-service","ipv4":"172.29.92.136","port":2222}},{"key":"http.path","value":"/accounts/showAll","endpoint":
{"serviceName":"accounts-service","ipv4":"172.29.92.136","port":2222}},
{"key":"http.url","value":"http://172.29.92.136:2222/accounts/showAll","endpoint":{"serviceName":"accounts-
service","ipv4":"172.29.92.136","port":2222}}]},
{"traceId":"6681da9173280aa7","id":"2df8f006fa15179c","name":"http:/ebusinessaccount/findall","parentId":"6681da9173280aa7","timestam
p":1482212949792000,"duration":4906000,"annotations":[{"timestamp":1482212949792000,"value":"cs","endpoint":{"serviceName":"accounts-
service","ipv4":"172.29.92.136","port":2222}},{"timestamp":1482212953149000,"value":"sr","endpoint":{"serviceName":"ebusiness-
service","ipv4":"172.29.92.136","port":2223}},{"timestamp":1482212954512000,"value":"ss","endpoint":{"serviceName":"ebusiness-
service","ipv4":"172.29.92.136","port":2223}},{"timestamp":1482212954698000,"value":"cr","endpoint":{"serviceName":"accounts-
service","ipv4":"172.29.92.136","port":2222}}],"binaryAnnotations":[{"key":"sa","value":true,"endpoint":{"serviceName":"accounts-
service","ipv4":"172.29.92.136","port":2222}}]},
{"traceId":"6681da9173280aa7","id":"4a86cb7481a982b0","name":"http:/ebusinessaccount/findall","parentId":"2df8f006fa15179c","timestam
p":1482212953147000,"duration":1366000,"binaryAnnotations":[{"key":"http.host","value":"172.29.92.136","endpoint":
{"serviceName":"ebusiness-service","ipv4":"172.29.92.136","port":2223}},{"key":"http.method","value":"GET","endpoint":
{"serviceName":"ebusiness-service","ipv4":"172.29.92.136","port":2223}},
{"key":"http.path","value":"/eBusinessAccount/findAll","endpoint":{"serviceName":"ebusiness-
service","ipv4":"172.29.92.136","port":2223}},
{"key":"http.url","value":"http://172.29.92.136:2223/eBusinessAccount/findAll","endpoint":{"serviceName":"ebusiness-
service","ipv4":"172.29.92.136","port":2223}},{"key":"lc","value":"unknown","endpoint":{"serviceName":"ebusiness-
service","ipv4":"172.29.92.136","port":2223}}]}]
```

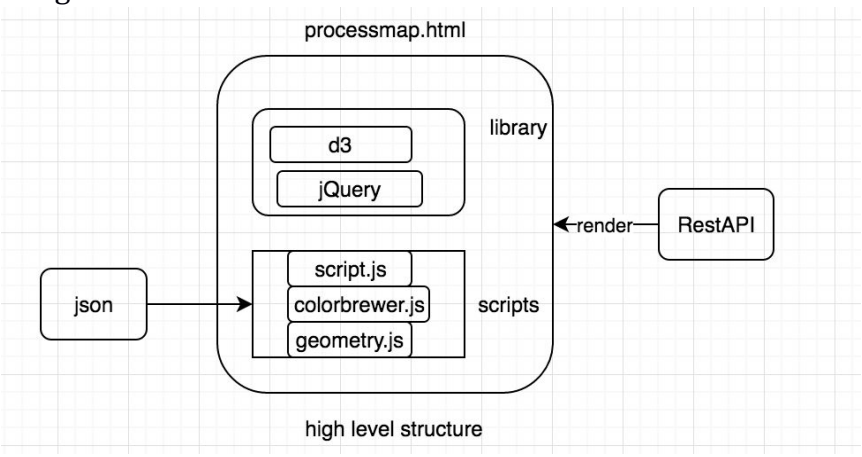

5.3.2 D3 Display

Like we mentioned in the previous chapter, d3 is used in the frontend dynamically loading the SVG and interactively present the contents we care about. We build a standalone microservice to display a single trace via html. The java source code are in the src/main/java directory.



The resources contains the template folder. We put all the html files in the folder. the We put the other resource file in the static file. We create some folders to hold some specific type of files.

Below is the high level structure of this microservice:



The REST API will render the process.html page when the corresponding web point is visited. The html include some common libraries like Jquery and d3. The script.js file is responsible for loading Span Data from the Collector, either from Zipkin Collector or our Customized Collector. Colorbrewer.js file is for coloration, geometry.js is for computing the coordinates in the graph to avoid conflicts.

There're hundreds of lines code in the script, for the simplicity of learning, some important snippets in script.js are illustrated below:

```
function getValues(traceId) {
    var result = {};
  $.ajax('http://localhost:9411/api/v1/trace/' + traceId, {
  type: 'GET',
  async: 'false',
  dataType: 'json'
    }).then(data => {
        for (var i in data) {
            var id = data[i].id;
            result[id] = data[i];
        }
        console.log(data);
        console.log(result);
        graph.data = result;
        drawGraph();

    });
}
```

Ajax call to load json file into graph.data for d3.

```
graph.links = [];
for (var name in graph.data) {
    var obj = graph.data[name];
    console.log(graph.data);
    if (obj.parentId != undefined) {
        console.log(obj.parentId);
        console.log(graph.data[obj.parentId]);
        var link = {
              source : graph.data[obj.parentId],
              target : obj
          };
        link.strength = 3;
         graph.links.push(link);
    }

}
```

Process the links between the Span Data;

```
graph.categories = {};
for (var name in graph.data) {
    var obj = graph.data[name],
        key = obj.type + ':' + (obj.group || ''),
        cat = graph.categories[key];
   if (obj.parentId == undefined) {
        key = 'trace Start';
   } else if (obj.annotations != undefined ) {
        key = 'endpoint';
   } else {
        key = 'local component';
   }
    obj.categoryKey = key;
    if (!cat) {
        cat = graph.categories[key] = {
            key       : key,
            type      : null ,//obj.type,
            type      : key,
            group     : key, //obj.group,
            count     : 0
        };
    }
    cat.count++;
}
```

Categorize the Span nodes.

6. Tutorial & Experiments & Analysis

There are basically two ways to run the applications: via command line or via IDE. We encourage to use the STS(Spring Tool Suite) developed by Pivotal to run the Spring Applications we have demonstrated in this report, which is far more convenient than running in the command line in most cases. But we will talk about both ways in the following part.

6.1 Distributed Tracing

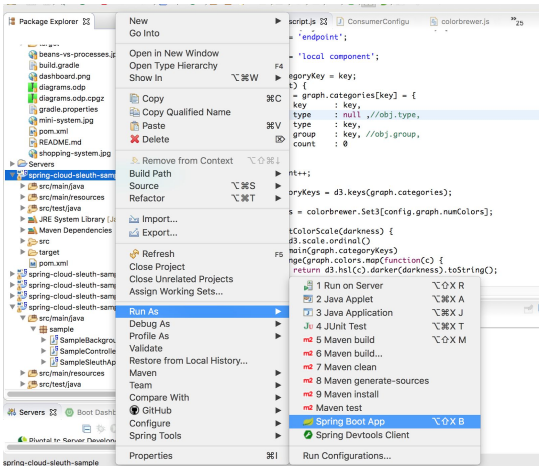
6.1.1 Sleuth(core) experiments

In this part, we will run the spring-cloud-sleuth-sample via STS to demonstrate Section 5.2.1.

(1) import project into STS

download code from https://github.com/spring-cloud/spring-cloud-sleuth using git clone or directly download and unzip. Then import into the STS.

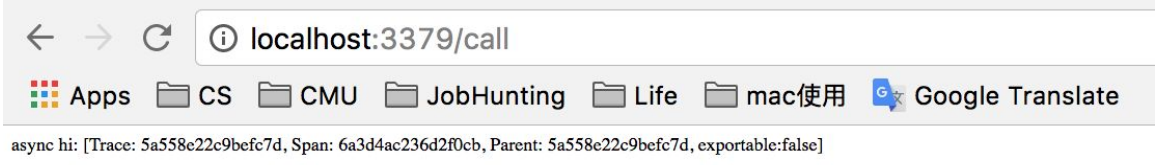
(2)run the application in STS
right click onto the project->Run As->Spring Boot App




```
spring-cloud-sleuth-sample - SampleSleuthApplication [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk1.7.0_79.jdk/Contents/Home/bin/java (Dec 20, 2016, 1:31:44 AM
[testSleuthApp,,,] 26772 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.cloud.sleuth.in
[testSleuthApp,,,] 26772 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 3379 (ht
[testSleuthApp,,,] 26772 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat
[testSleuthApp,,,] 26772 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicati
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initializatio
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter'
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  : Mapping filter: 'traceFilter' to: [/*]
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter'
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to
[testSleuthApp,,,] 26772 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.spring
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/start]}" onto public java.lang
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/call]}" onto public java.util.
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/async]}" onto public java.lang
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/]}" onto public java.lang.Stri
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/hi2]}" onto public java.lang.S
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/traced]}" onto public java.lan
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.sprin
[testSleuthApp,,,] 26772 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}"
[testSleuthApp,,,] 26772 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handle
[testSleuthApp,,,] 26772 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of typ
[testSleuthApp,,,] 26772 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto ha
[testSleuthApp,,,] 26772 --- [           main] o.s.s.c.ThreadPoolTaskScheduler          : Initializing ExecutorService
[testSleuthApp,,,] 26772 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on sta
[testSleuthApp,,,] 26772 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 3379 (http)
[testSleuthApp,,,] 26772 --- [           main] sample.SampleSleuthApplication           : Started SampleSleuthApplication in 10.711
```

Then you can see the output in the console in STS. If you run the project in the console, you should see the similar information there.

(3) Test the API by visiting localhost:3379/call
Open Browser to visit localhost:3379/call



The info printed in the web page is the trace file generated. And the "exportable:false" means the data has not been exported to remote service or message queue.

Note that: the default port is 3379, you can modify the port in the yml file;Note that: you can also check the web point specified in the controller.java file, like /async..

6.1.2 Sleuth Stream + RabbitMQ + Zipkin

In this part, we will run the **spring-cloud-sleuth-sample-stream** via STS to demonstrate Section 5.2.2.
(1) Import project into STS
download code from https://github.com/spring-cloud/spring-cloud-sleuth using git clone or directly download and unzip. Then import into the STS.

(2) Install the RabbitMQ and run RabbitMQ server
Note that:This step should be run before (3)(4)!!
Reference for MAC system:
https://www.rabbitmq.com/install-standalone-mac.html

Install the Server

Before installing make sure you have the latest brews:

```
brew update
```

Then, install RabbitMQ server with:

```
brew install rabbitmq
```

Run RabbitMQ Server

The RabbitMQ server scripts are installed into `/usr/local/sbin`. This is not automatically added to your path, so you may wish to add `PATH=$PATH:/usr/local/sbin` to your `.bash_profile` or `.profile`. The server can then be started with `rabbitmq-server`.

All scripts run under your own user account. Sudo is not required.

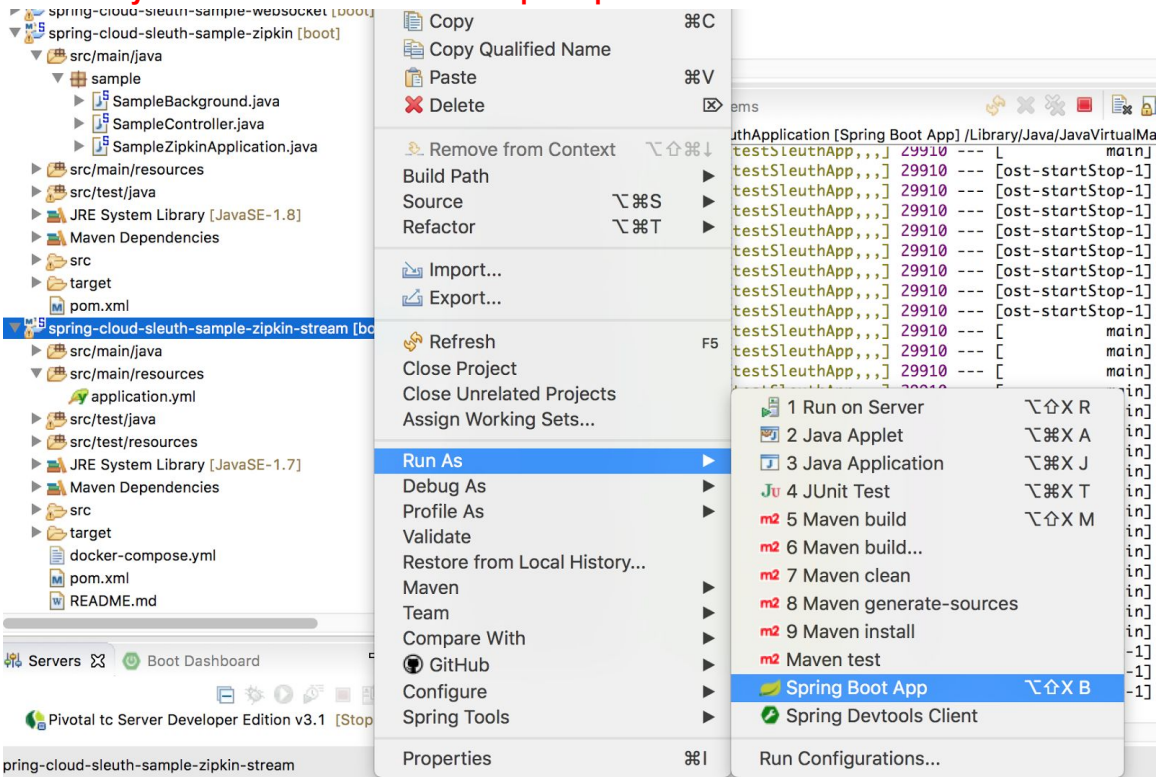
Then Run: **rabbitmq-server start** in the console as in the screenshot below
Then you should see the results like the screenshot:

```
➜  ~ git:(master) ✗ rabbitmq-server start

              RabbitMQ 3.6.4. Copyright (C) 2007-2016 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: /usr/local/var/log/rabbitmq/rabbit@localhost.log
  ######  ##        /usr/local/var/log/rabbitmq/rabbit@localhost-sasl.log
  ##########
              Starting broker...
 completed with 10 plugins.
```

Note that: you might need to use Source command if rabbit-server command could not be found when executing “rabbitmq-server start”;

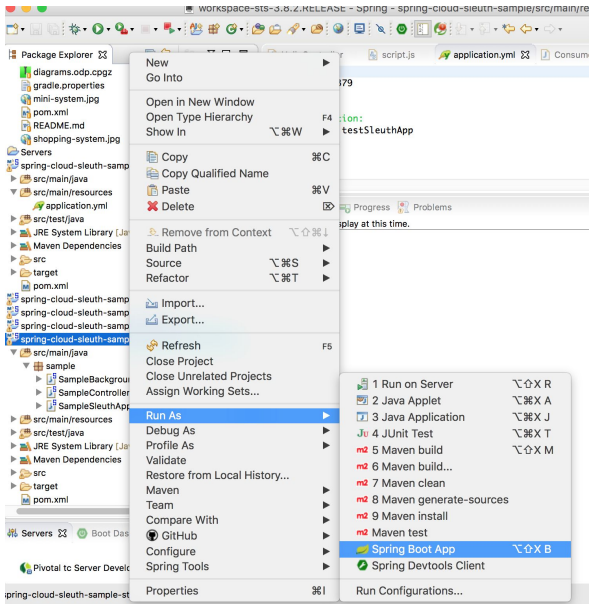
(3) run the embedded zipkin project :`spring-cloud-sleuth-sample-stream-zipkin`

Note that: you have to use JAVA8 to compile zipkin related file

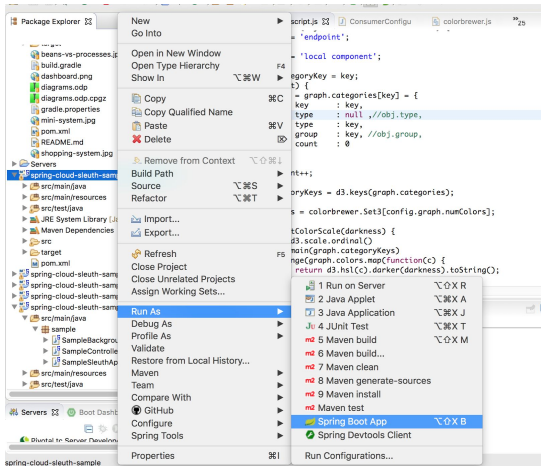


then you should see the service started in the console

(4)run the Span Data application in STS: Spring-cloud-sleuth-sample-stream. Since we have zipkin and RabbitMQ running, we can start the application now. right click onto the project->Run As->Spring Boot App



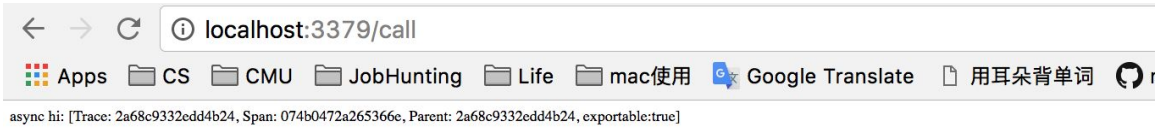
then you will see the similar info in the STS console.



```
i.j.e.a.AnnotationMBeanExporter          : Bean with name 'rabbitConnectionFactory' has been autodetected for JMX
i.j.e.a.AnnotationMBeanExporter          : Located managed bean 'integrationMbeanExporter': registering with JMX s
i.j.e.a.AnnotationMBeanExporter          : Located managed bean 'rabbitConnectionFactory': registering with JMX se
i.i.monitor.IntegrationMBeanExporter     : Registering beans for JMX exposure on startup
i.i.monitor.IntegrationMBeanExporter     : Registering MessageChannel errorChannel
i.i.monitor.IntegrationMBeanExporter     : Located managed bean 'org.springframework.integration:type=MessageChann
i.i.monitor.IntegrationMBeanExporter     : Registering MessageChannel nullChannel
i.i.monitor.IntegrationMBeanExporter     : Located managed bean 'org.springframework.integration:type=MessageChann
i.i.monitor.IntegrationMBeanExporter     : Registering MessageChannel sleuth
i.i.monitor.IntegrationMBeanExporter     : Located managed bean 'org.springframework.integration:type=MessageChann
i.i.monitor.IntegrationMBeanExporter     : Located managed bean 'org.springframework.integration:type=MessageHandl
i.i.monitor.IntegrationMBeanExporter     : Located managed bean 'org.springframework.integration:type=MessageSourc
i.c.support.DefaultLifecycleProcessor    : Starting beans in phase -2147482648
mple.SampleSleuthApplication             : No active profile set, falling back to default profiles: default
:.a.AnnotationConfigApplicationContext   : Refreshing org.springframework.context.annotation.AnnotationConfigAppli
i.c.support.GenericApplicationContext    : Refreshing org.springframework.context.support.GenericApplicationContex
mple.SampleSleuthApplication             : Started SampleSleuthApplication in 0.227 seconds (JVM running for 13.80
i.a.r.c.CachingConnectionFactory         : Created new connection: SimpleConnection@952b025 [delegate=amqp://guest
i.integration.channel.DirectChannel      : Channel 'testSleuthApp:3379.sleuth' has 1 subscriber(s).
i.c.support.DefaultLifecycleProcessor    : Starting beans in phase 0
i.i.endpoint.EventDrivenConsumer         : Adding {logging-channel-adapter:_org.springframework.integration.errorL
i.i.channel.PublishSubscribeChannel      : Channel 'testSleuthApp:3379.errorChannel' has 1 subscriber(s).
i.i.endpoint.EventDrivenConsumer         : started _org.springframework.integration.errorLogger
i.i.e.SourcePollingChannelAdapter        : started sleuthStreamSpanReporter.poll.inboundChannelAdapter
i.c.support.DefaultLifecycleProcessor    : Starting beans in phase 2147482647
i.c.support.DefaultLifecycleProcessor    : Starting beans in phase 2147483647
).c.e.t.TomcatEmbeddedServletContainer   : Tomcat started on port(s): 3379 (http)
mple.SampleSleuthApplication             : Started SampleSleuthApplication in 13.344 seconds (JVM running for 14.2
```

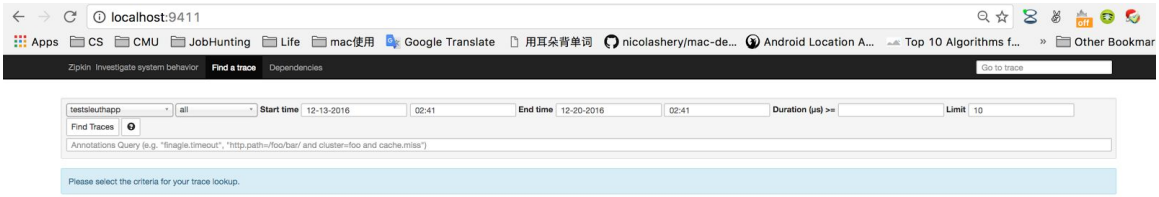
Then you can see the output in the console in STS. If you run the project in the console, you should see the similar information there.

(3) Test the API by visiting localhost:3379/call
Open Browser to visit localhost:3379/call



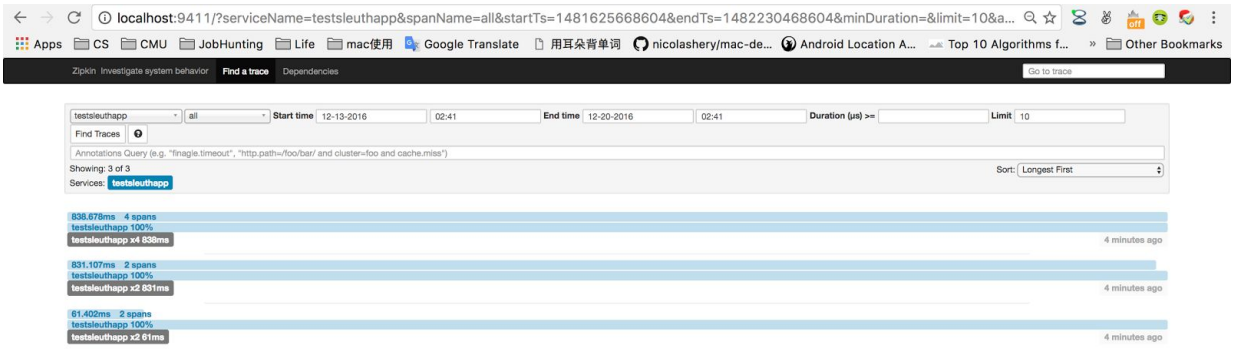
As you can see in the webpage, we get a Span Data. And now the “exportable:true”, which means the data has been successfully sent to RabbitMQ. Also, you can test other APIs if you wish.

(4) Test the Zipkin
Visit 9411 port in the browser to check Zipkin.

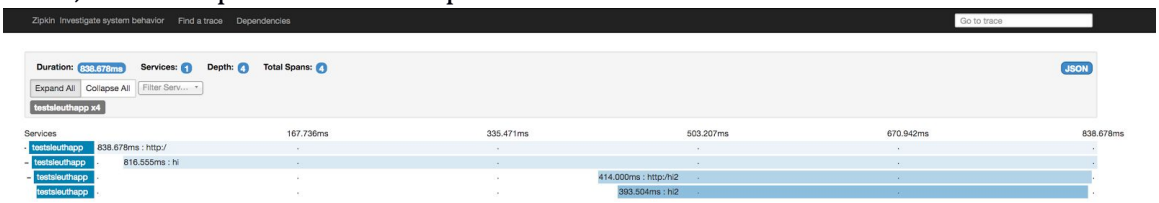


As you can see from the page, zipkin is up and running.

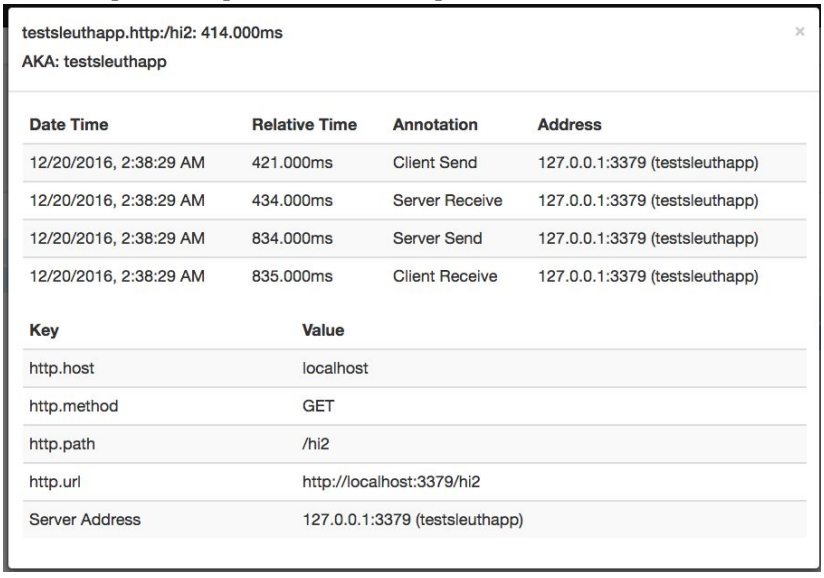
(5) choose a proper start time and end time, and click “find traces” button to filter the span Data. We can see from the Screenshot below, zipkin has captured 3 traces.



If we click the first one, we can check for more details, like the duration, service name, timestamp of the certain Span.



We can choose the specific Span to see the Span Data:



In addition, the zipkin provide the json format of the current Trace(all the Spans shown are of the same trace) .

6.1.3 Sleuth Stream + RabbitMQ + Customized Consumer

In this part, we will run the **spring-cloud-sleuth-sample-stream** via STS to demonstrate Section 5.2.3. The (1)(2) are the same with 6.1.2.

(1) Import project into STS

download code from https://github.com/spring-cloud/spring-cloud-sleuth using git clone or directly download and unzip. Then import into the STS.

(2) Install the RabbitMQ and run RabbitMQ server

Note that:This step should be run before (3)(4)!!

Reference for MAC system:

https://www.rabbitmq.com/install-standalone-mac.html

Install the Server

Before installing make sure you have the latest brews:

```
brew update
```

Then, install RabbitMQ server with:

```
brew install rabbitmq
```

Run RabbitMQ Server

The RabbitMQ server scripts are installed into `/usr/local/sbin`. This is not automatically added to your path, so you may wish to add `PATH=$PATH:/usr/local/sbin` to your `.bash_profile` or `.profile`. The server can then be started with `rabbitmq-server`.

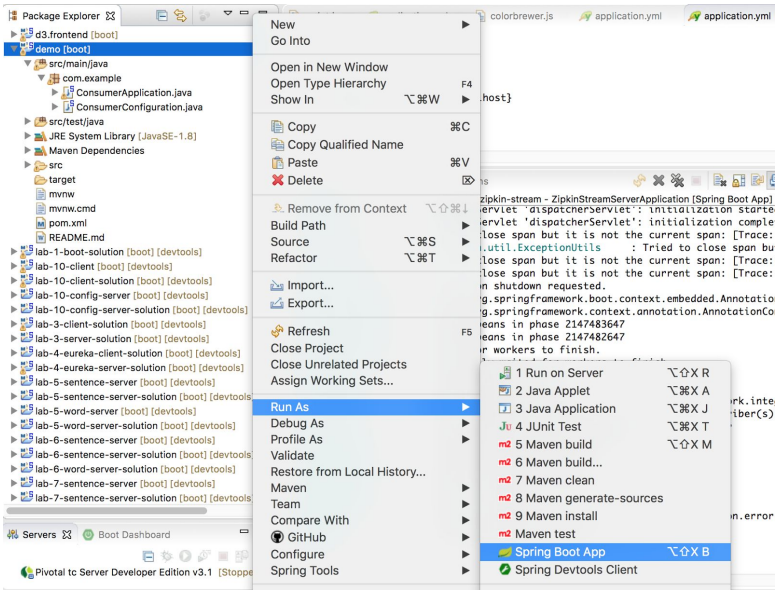
All scripts run under your own user account. Sudo is not required.

Then Run: **rabbitmq-server start** in the console as in the screenshot below
Then you should see the results like the screenshot:

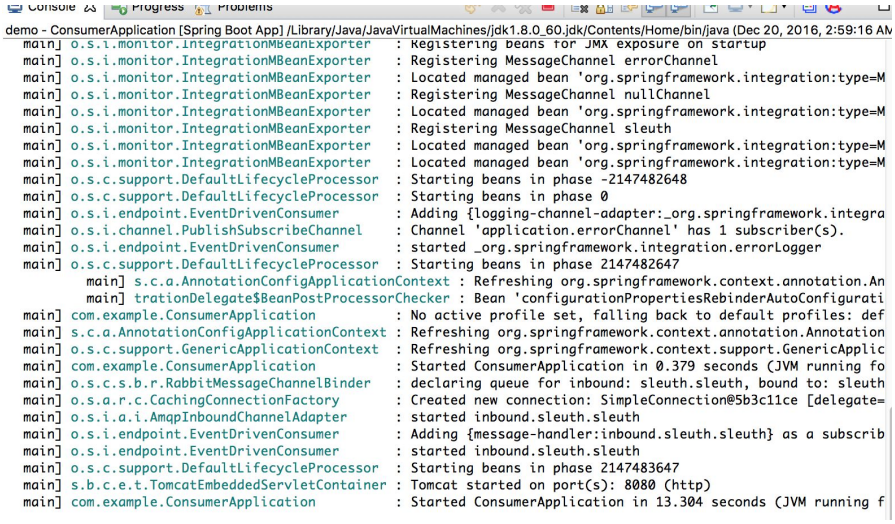
```
➜  ~ git:(master) ✗ rabbitmq-server start

              RabbitMQ 3.6.4. Copyright (C) 2007-2016 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: /usr/local/var/log/rabbitmq/rabbit@localhost.log
  ######  ##        /usr/local/var/log/rabbitmq/rabbit@localhost-sasl.log
  ##########
              Starting broker...
 completed with 10 plugins.
```

Note that: you might need to use Source command if rabbit-server command could not be found when executing “rabbitmq-server start”;

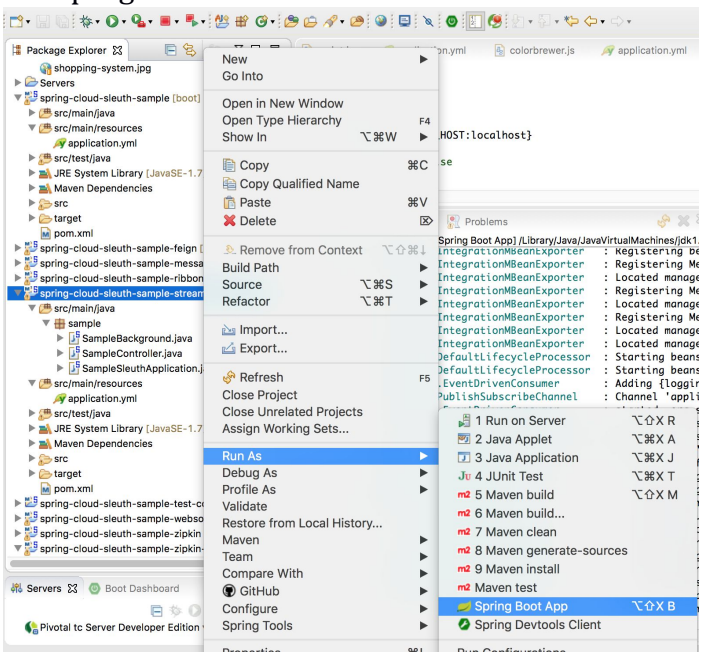
(3)Run the custom consumer program: demo
Note that: if you have multiple projects named demos, you can choose to rename this project before loading into STS or delete other projects temporarily from workspace.



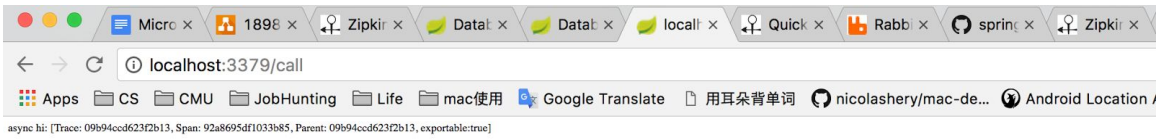
The custom consumer is up and running:



(4) Run the producer program:



(5) check the APIs



Note that: the "exportable:true", which means the Span Data have been sent to message queue.

(6) check the console of custom customer:

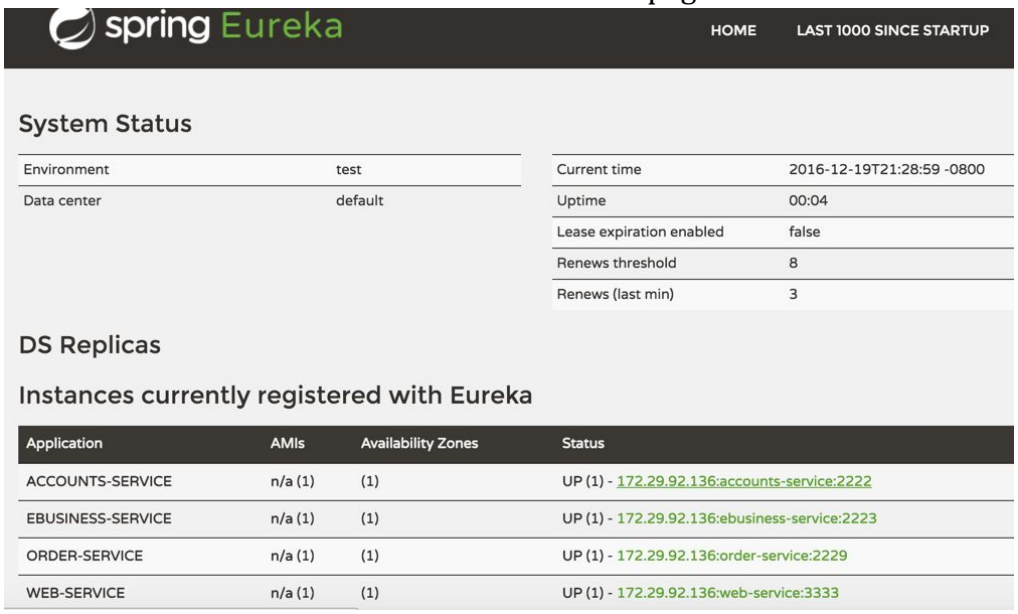
The Span data has been successfully logged in the console :

```
.7
http)
.304 seconds (JVM running for 14.137)
   : In consumer: the Spans are: [[Trace: 9b94ccd623f2b13, Span: 92a8695df1033b85, Parent: 9b94ccd623f2b13, exportable:true]]
   : In consumer: the Spans are: [[Trace: 9b94ccd623f2b13, Span: 9b94ccd623f2b13, Parent: null, exportable:true]]
```

6.1.4 Sleuth + Zipkin Collector(Customized Consumer) + D3

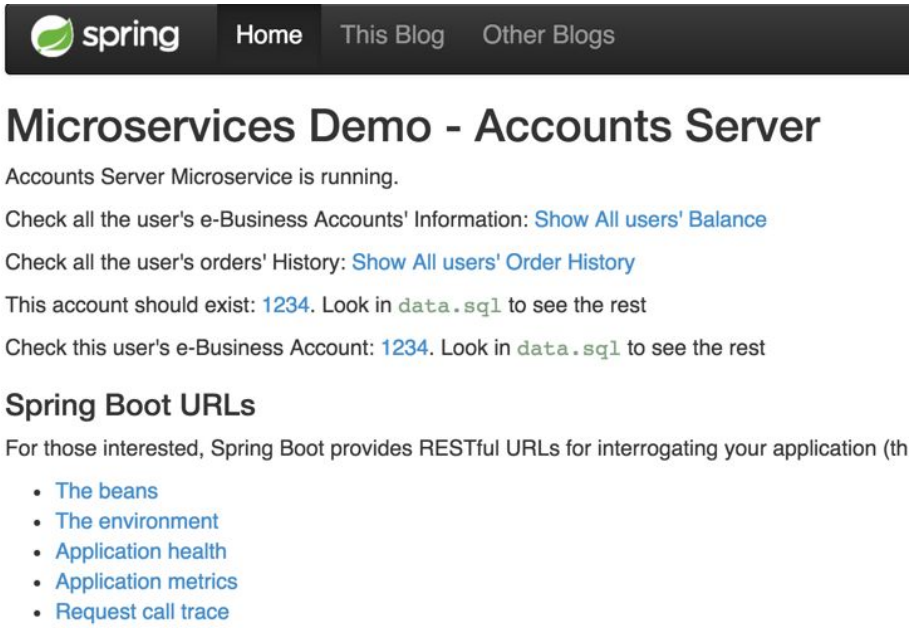
a. Get the Billing microservices system start running

- Firstly we need to get the microservices running. In Billing Microservices system project folder, run "mvn clean install" or "mvn clean install -DskipTests", dependencies will be installed and microservices project will be compiled.
- Get each microservices running by invoking the microservice-BillingSystem-1.1.0.RELEASE.jar file with different input arguments: "reg" is for registration service- Eureka; "ebus" is for e-Business service; "accounts" is for Account service; "order" is for MyOrder service; "web" is for Web Service. After all the services get start running, one can check the status of the services in the Eureka page:

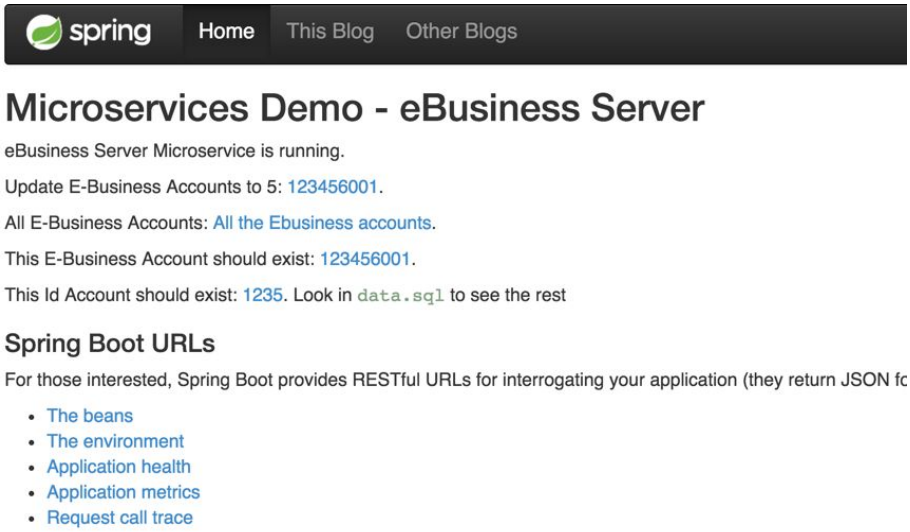


this shows that all the services have been started successfully!

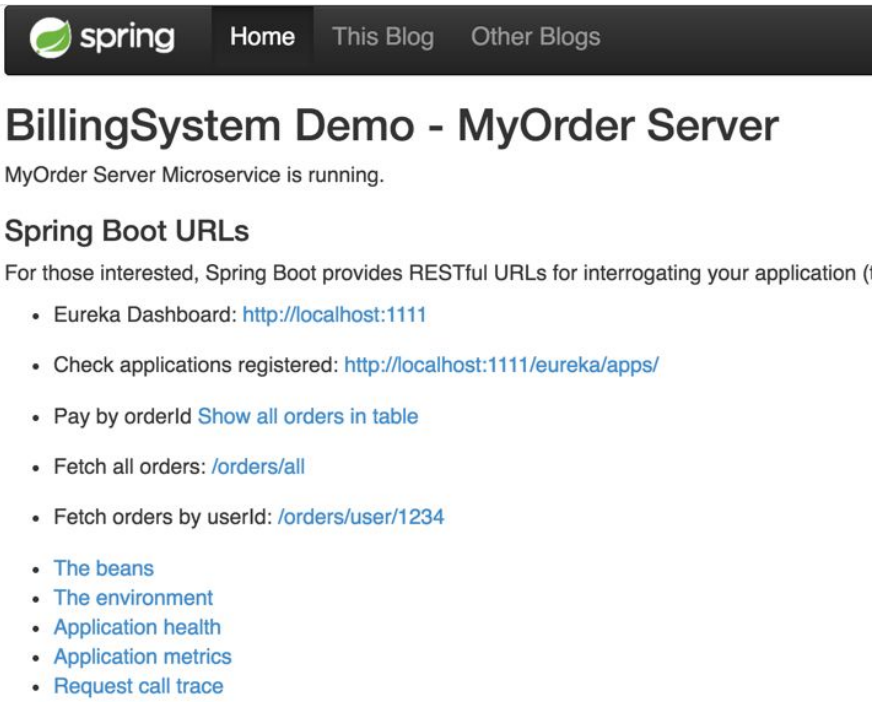
- Go to "localhost:2222" to check ACCOUNTS-SERVICE's UI



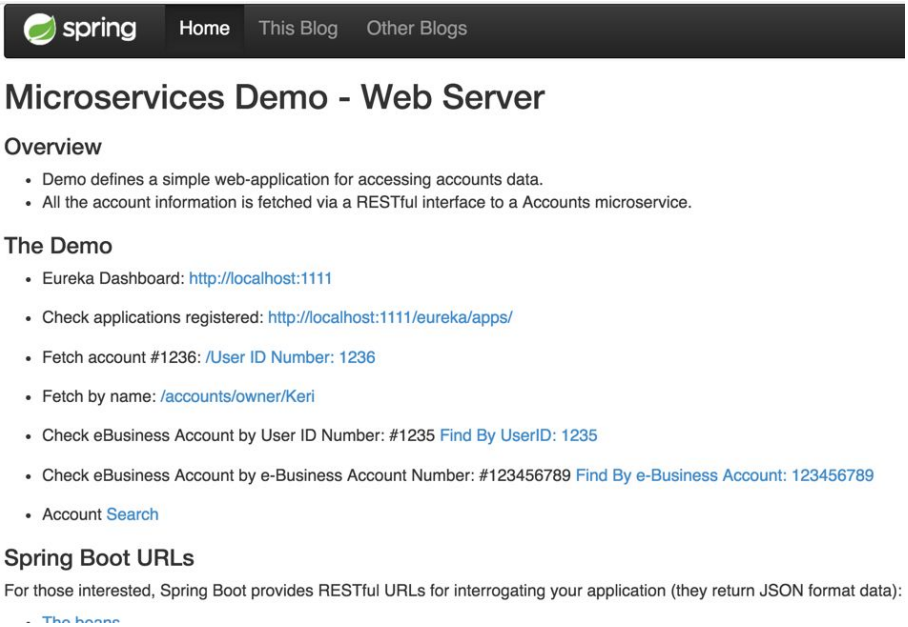
- Go to "localhost:2223" to check EBUSINESS-SERVICE's UI



- Go to "localhost:2229" to check ORDER-SERVICE's UI



- Go to "localhost: 3333" to check WEB-SERVICE's UI



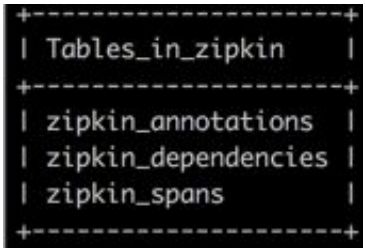
b. Configure local database

Before running zipkin we need to setup the local database.

- Get into the Zipkin project folder and run the database schema to create the correct local database "zipkin" and corresponding tables in the database. Under the Zipkin folder, run:
- `$ mysql -uroot -e "SET GLOBAL innodb_file_format=Barracuda"`

- `$ mysql -uroot -e "show global variables like 'innodb_file_format'"`
- `$ mysql -uroot -e "create database if not exists zipkin"`
- `$ mysql -uroot -Dzipkin < zipkin-storage/mysql/src/main/resources/mysql.sql -p`

After this check local MySQL database: there will be a database named “zipkin” with tables inside:



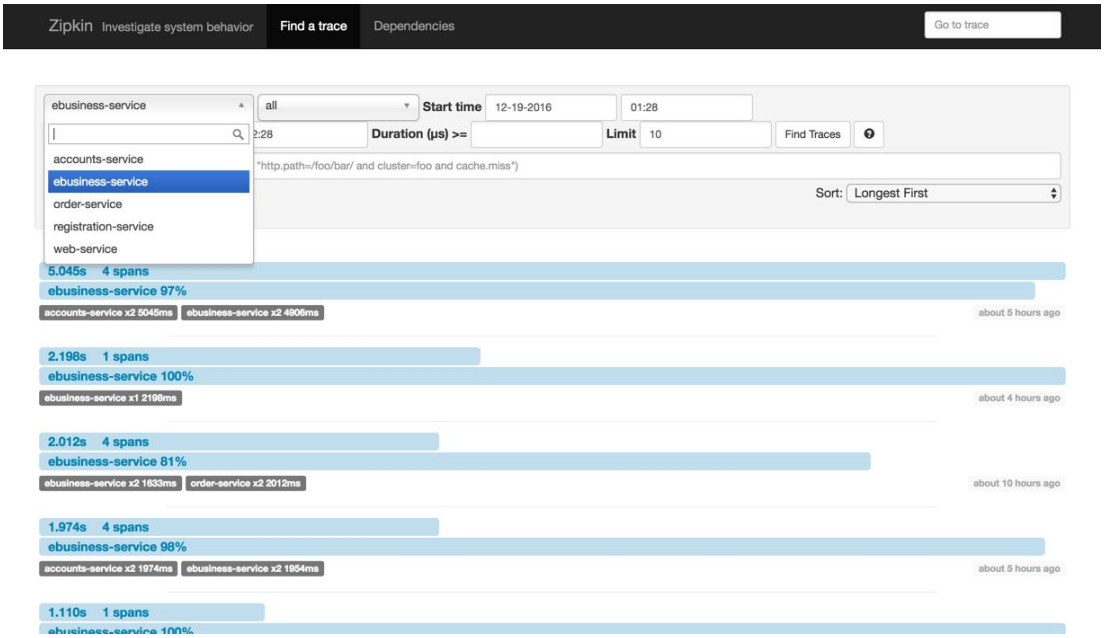
This means the local database has been setup successfully.

c. Get Zipkin service start running

zipkin can be accessed via docker image, source files or jar file. We choose to use the zipkin.jar. Note that: to run Zipkin, you have to have JAVA8 installed. There are three ways to download zipkin, you can refer to:
http://zipkin.io/pages/quickstart.html

After the above steps, we need to run the zipkin server.

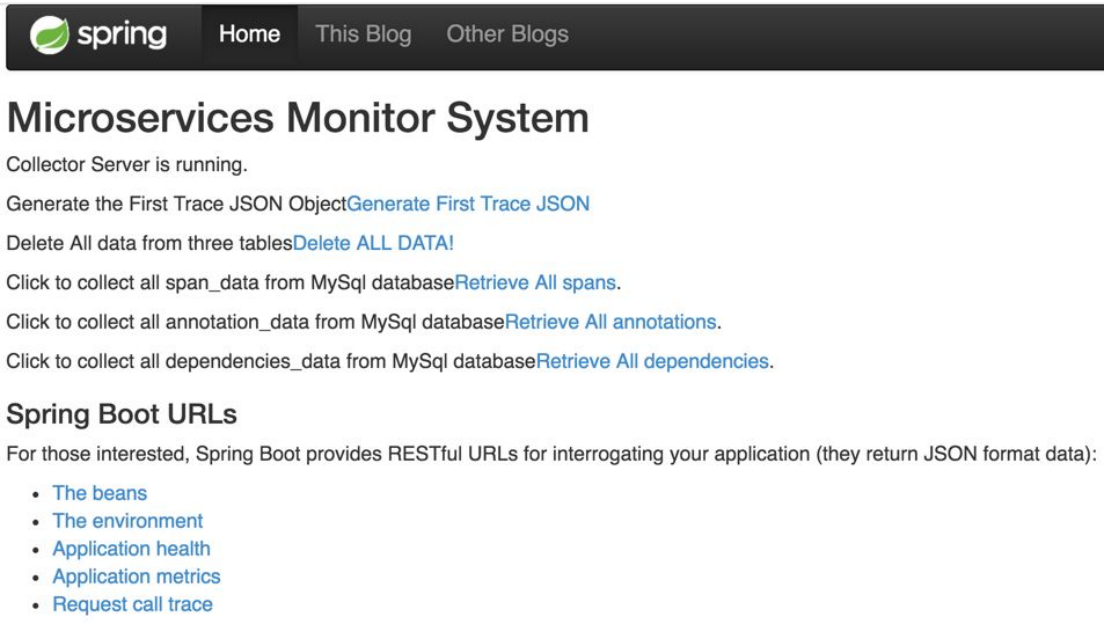
- Under the folder of zipkin.jar, run this: “STORAGE_TYPE=mysql MYSQL_USER=root MYSQL_PASS=root java -jar zipkin.jar”
- Open “localhost:9411” and zipkin UI will be showing.
- Select the correct service and appropriate time range, the spans of the corresponding service will be showing:



d. Get the Collector-Service running

To collect the services spans and traces and convert them into JSON file, one need to run the Collector Service.

- Under the "Collector-Service" folder, run "mvn clean install -DskipTests", project will be installed and compiled.
- run "java -jar target/Collector-Service-1.1.0.RELEASE.jar col".
- Go to "localhost:2225" to check the "Collector-Service" UI



- Test the Collector by click on the first link. The generated JSON trace will be returned:

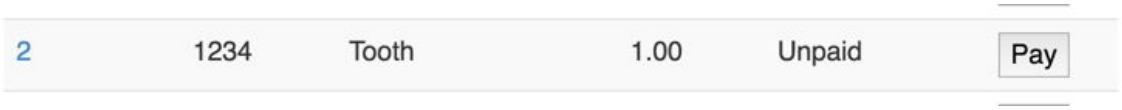
```
[{"traceId":"2458899367954344141","duration":"2000","binaryAnnotations":[{"endpoint":
{"ipv4":"127.0.0.1","port":"1111","serviceName":"registration-
service"},"value":"localhost","key":"http.host"},{"endpoint":
{"ipv4":"127.0.0.1","port":"1111","serviceName":"registration-service"},"value":"PUT","key":"http.method"},
{"endpoint":{"ipv4":"127.0.0.1","port":"1111","serviceName":"registration-
service"},"value":"\/eureka\/apps\/EBUSINESS-SERVICE\/172.29.92.136:ebusiness-
service:2223","key":"http.path"},{"endpoint":{"ipv4":"127.0.0.1","port":"1111","serviceName":"registration-
service"},"value":"http:\/\/localhost:1111\/eureka\/apps\/EBUSINESS-SERVICE\/172.29.92.136:ebusiness-
service:2223?
status=UP&lastDirtyTimestamp=1482192112377","key":"http.url"}],"name":"http:\/eureka\/apps\/ebusiness-
service\/172.29.92.136:ebusiness-service:2223","annotations":[{"endpoint":
{"ipv4":"127.0.0.1","port":"1111","serviceName":"registration-
service"},"value":"sr","timestamp":"1482192172148000"},{"endpoint":
{"ipv4":"127.0.0.1","port":"1111","serviceName":"registration-
service"},"value":"ss","timestamp":"1482192172150000"}],"id":"2458899367954344141","timestamp":"148219217214
8000"}]
```


And a "test.json" file will also be generated in the project folder under the root path. This JSON file can be used for future analysis or front-end displaying.

e. Get the D3 service start running

To display the services' APIs requests process, we need to get the D3 service start running.

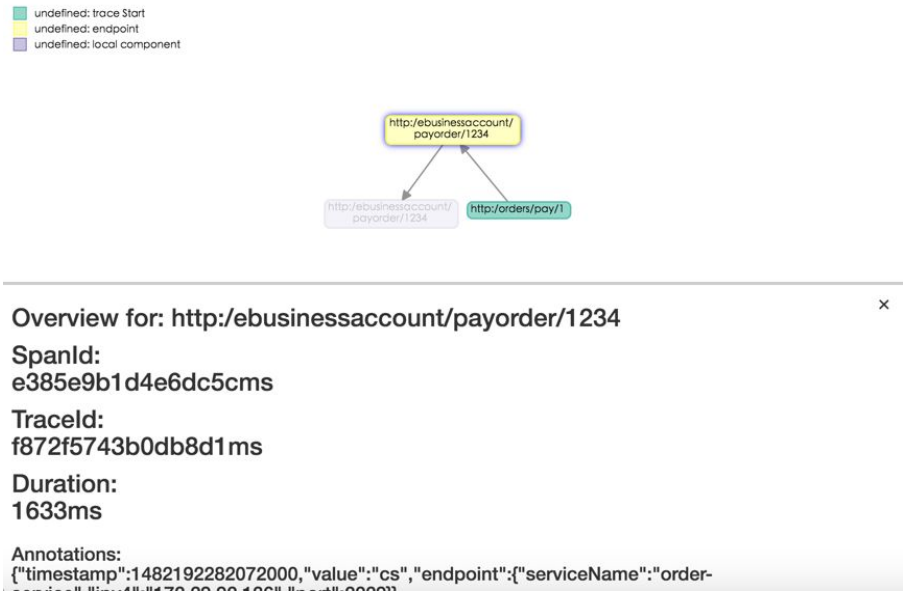
- Under the d3.frontend folder, run "mvn clean install -DskipTests"
- After the project being compiled, run "java -jar target/d3.frontend-0.0.1-SNAPSHOT.jar"
- Go to "localhost: 3300" and you will see the service is running.
- To test the D3 display function, first we need to create API calls between services. Here we did a "paying order" operation: From MyOrder service we open the orders table and choose to pay a "tooth" order. A "PUT" request will be sent to e-Business service and



e-Business service will receive this order and process with the back-end database.

- After creating the above scenario, use api "http://localhost:3300/d3processmap" to query the d3 service and ask for generate the map of the scenario, and you will see the demo Map of the above scenario as shown below

- If you click on certain node, the detailed information will be shown at the bottom of the page.



We can clearly see the duration, and annotations from the html page, whose information could be a very useful supplement to Zipkin visualization.

7.Conclusions and Thoughts

Result:

Firstly, we have carefully designed and developed our own mini-billing systems based on microservices with Spring-Cloud. This system is low coupling and high cohesion, which is implemented by modularity implementations and HTTP calls among the microservices. The system is made up five micro services, both with frontend and backend, and most of them has individual databases.

Secondly, we have investigated the explored four ways of the usages of Spring Cloud Sleuth, along with some technologies with Zipkin, RabbitMQ:1)Sleuth-core;(2)Sleuth-Stream + RabbitMQ + Zipkin; (3)Sleuth Stream + RabbitMQ + customized consumer; (4) Sleuth + Zipkin.

Thirdly, Under Jia's guidance, we have implemented a working solution from producing Span Data, consuming Span Data and Displaying Span Data. We have successfully integrated the Spring-Cloud-Sleuth, Zipkin, and D3 into our mini-billing system. The Sleuth is used for producing the Span Data, Zipkin for Collecting the Span Data, D3 for displaying the D3 data. In addition, we wrote a custom Span Data Collector. All of these are working as expected.

Thoughts:

First, Open Source Projects are "One for all, All for One". Thanks the open source coder who have helped us and have built the existing framework for us. (1)We

should stand on the shoulder of giants, leveraging the existing work to develop our own. (2) We can learn more by involving in the communication in the community. (3) Of course, our work still has some problems, and any suggestions will be appreciated.

Second, we need more APIs to test our work. Since we only have five microservices right now, and the Span Data depth is only 2 or 3. If we have more APIs, the result of Distributed Tracing would be more complex and interesting.

Third, Improvement with D3 microservices. The D3 part we have built can only show a single trace data at one time. It would be more interesting to see multiple trace data together in a single graph, and some powerful improved D3 framework could be used in this project to improve the UI.

Last but not Least, we have learnt a lot from this course. We could not have achieved this without the guidance of Jia. Thanks to her.

8. Contribution of each team member

Fei Xu: 50% contribution to the whole project
1) e-Business microservice (frontend, backend), Web microservice frontend and backend APIs, Account microservice frontend and backend APIs,
2) Troubleshooting and bugs fixing in APIs communications between services.
3) Experiments and Configuration with Sleuth, Zipkin(Pair Programming with Yichen Lin)
4) Collector Service for MySQL data retrieving and translating.
5) Co-author the report with Yichen Lin

Yichen Lin: 50% contribution to the whole project
1)MyOrder micro-service (frontend, backend)
2) Troubleshooting and bugs fixing
3) Experiments and Configuration with Sleuth, Zipkin(Pair Programming with Fei XU)
4) D3 visualization microservice
5) Co-author this report with Fei XU.

Appendix:

-Check in everything onto GitHub:
https://github.com/xfeifly/MicroServices-Billing.git

References

[1]https://cloud.spring.io/spring-cloud-sleuth/
[2]Dapper:https://research.google.com/pubs/pub36356.html
[3]https://en.wikipedia.org/wiki/RabbitMQ
[4]Zipkin at Twitter: https://www.youtube.com/watch?v=EP4prWJIWvc
[5]D3:https://d3js.org/
[6]Sleuthtutorial:https://cloud.spring.io/spring-cloud-sleuth/spring-cloud-sleuth.html#_span_data_as_messages
[7]zipkin download page: http://zipkin.io/pages/quickstart.html
[8] Spring Cloud Sleuth: https://github.com/spring-cloud/spring-cloud-sleuth.git
[9] Zipkin: https://github.com/openzipkin/zipkin.git
[10] Microservices: https://github.com/paulc4/microservices-demo.git