Bipin Kumar

Shael.dhn88@gmail.com

Sleuth and zipkin

Sleuth and Zipkin Setup Doc

Distributed tracing using sleuth and zipkin:

Let assume we have order-service whenever a request come it will generate order id and logs it to log and after that it will hit other service like S1, S2 and S3. But in those we don’t have order id logged into logs. As we have already seen that we can configure ELK to centralize logs. We can see the logs. But suppose there are multiple order request are coming at same time and we want to trace only a particular order request. In case of distributed system, it will be difficult. But spring has provided a library called sleuth by which we can achieve this. We need to just add it to every service. There will be autoconfigure which will configure request interceptor. What it will do it will generate trace id and span id to trace it which is stored in thread local object. Now when a request come to book an order what it will do it will call other service in this case sleuth also auto configure rest template interceptor. Here what rest template will do it will send trace id as part of request. So, in case of call from order-service it will send trace id in request to S1. In S1 we have MVC interceptor which will check if trace id is there then don’t generate new trace id. But it will always generate new span id. Now we can go to kibana and search the logs and get the trace id and using trace id we can filter out all the logs. For this we don’t have to do any configuration except in logback.xml we need add trace id and span id.

1. Add below dependency in you project.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

1. For making it simple we only configuring console appender.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<configuration>

<appender name=*"STDOUT"* class=*"ch.qos.logback.core.ConsoleAppender"*>

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread, %X{X-B3-TraceId:-},%X{X-B3-SpanId:-}] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level=*"info"*>

<appender-ref ref=*"STDOUT"* />

</root>

</configuration>

1. Now just start eureka, cargo-booking and then cargo-routing application. At time of startup we can see trace id and span id is blank it is because it will be generated only in case if we make request.
2. Now open swagger-ui of cargo-booking service.

Open swagger web ui at <http://localhost:8081/swagger-ui.html>

Under Cargo Booking Commands in Swagger ui, make a post request to /cargobooking using the below json to create a booking.

{

"bookingAmount": 100,

"originLocation": "BLR",

"destLocation" : "DEL",

"destArrivalDeadline" : "2020-01-28"

}

Now observe logs you can see at startup of request span id and trace id is same.

19:10:57.189 [http-nio-8081-exec-8, 92929a50963ceb7c,92929a50963ceb7c] INFO c.j.b.i.rest.CargoBookingController - \*\*\*\*Cargo Booked \*\*\*\*100

Random is :572B4070-CEFE-4EEA-BDC2-A46A7FDF386D

Publishing CargoBookedEvent with booking id \*\*\* 572B4070-CEFE-4EEA-BDC2-A46A7FDF386D

19:10:57.463 [http-nio-8081-exec-8, 92929a50963ceb7c,8d199e0561f5f270] INFO o.s.a.r.c.CachingConnectionFactory - Attempting to connect to: [localhost:5672]

19:10:57.496 [http-nio-8081-exec-8, 92929a50963ceb7c,8d199e0561f5f270] INFO o.s.a.r.c.CachingConnectionFactory - Created new connection: rabbitConnectionFactory.publisher#54776f1c:0/SimpleConnection@69d3edc6 [delegate=amqp://guest@127.0.0.1:5672/, localPort= 58245]

Now use cargo-routing service and check trace id and span id.

Cargo-booking logs:

19:13:27.463 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.j.b.i.rest.CargoRoutingController - \*\*\*\*\*Inside CargoRoutingController.routeCargo\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CargoBookingCommandService.assignRouteToCargo()===========================

19:13:28.065 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.n.config.ChainedDynamicProperty - Flipping property: routing-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

19:13:28.105 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.n.u.c.ShutdownEnabledTimer - Shutdown hook installed for: NFLoadBalancer-PingTimer-routing-service

19:13:28.105 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.n.loadbalancer.BaseLoadBalancer - Client: routing-service instantiated a LoadBalancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=routing-service,current list of Servers=[],Load balancer stats=Zone stats: {},Server stats: []}ServerList:null

19:13:28.147 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.n.l.DynamicServerListLoadBalancer - Using serverListUpdater PollingServerListUpdater

19:13:28.205 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.n.config.ChainedDynamicProperty - Flipping property: routing-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

19:13:28.221 [http-nio-8081-exec-1, 1797279991b6ed92,1797279991b6ed92] INFO c.n.l.DynamicServerListLoadBalancer - DynamicServerListLoadBalancer for client routing-service initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=routing-service,current list of Servers=[localhost:8082],Load balancer stats=Zone stats: {defaultzone=[Zone:defaultzone; Instance count:1; Active connections count: 0; Circuit breaker tripped count: 0; Active connections per server: 0.0;]

},Server stats: [[Server:localhost:8082; Zone:defaultZone; Total Requests:0; Successive connection failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 05:30:00 IST 1970; First connection made: Thu Jan 01 05:30:00 IST 1970; Active Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]

]}ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList@3a0c6817

19:13:29.159 [PollingServerListUpdater-0, ,] INFO c.n.config.ChainedDynamicProperty - Flipping property: routing-service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

Publishing CargoRoutedEvent with booking id \*\*\*572B4070-CEFE-4EEA-BDC2-A46A7FDF386D

Routing logs:

19:13:28.635 [http-nio-8082-exec-1, ,] INFO o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing Spring DispatcherServlet 'dispatcherServlet'

19:13:28.635 [http-nio-8082-exec-1, ,] INFO o.s.web.servlet.DispatcherServlet - Initializing Servlet 'dispatcherServlet'

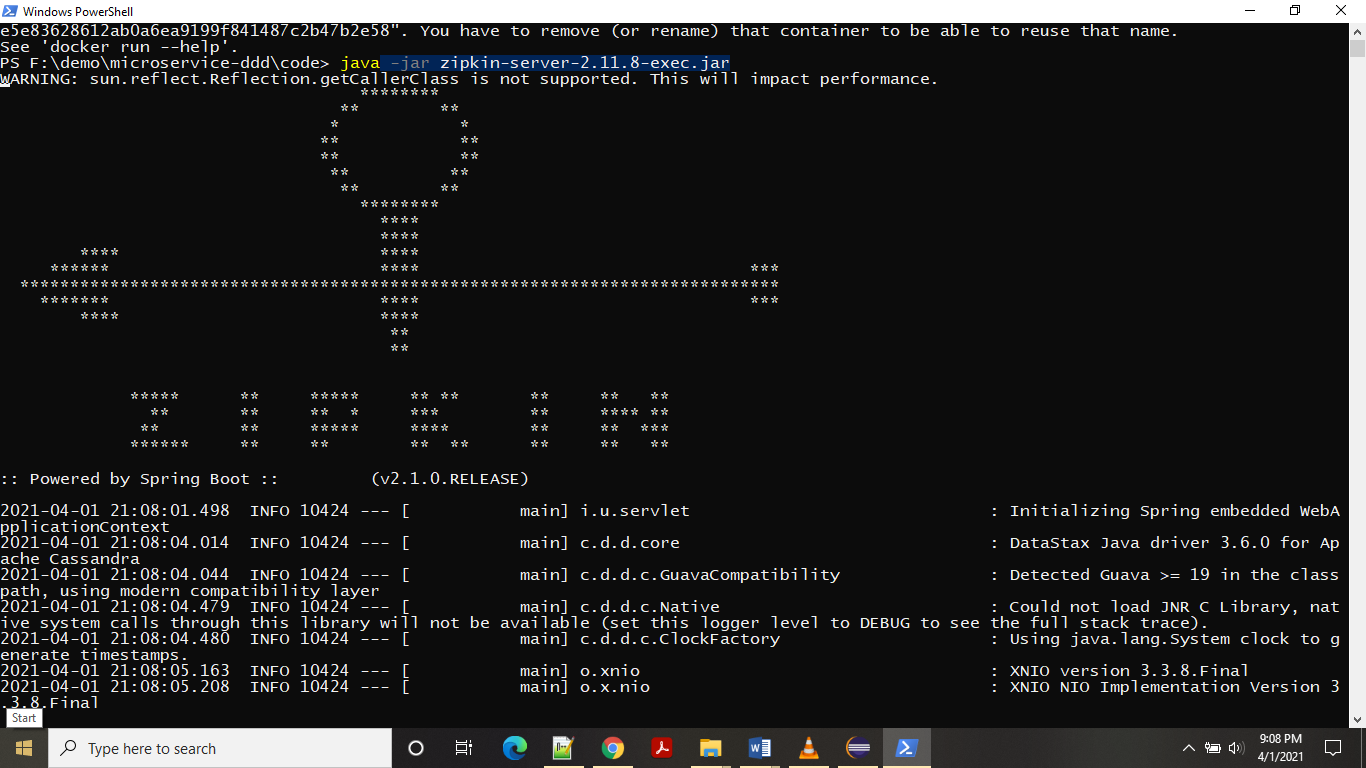
19:13:28.675 [http-nio-8082-exec-1, ,] INFO o.s.web.servlet.DispatcherServlet - Completed initialization in 40 ms

19:13:28.802 [http-nio-8082-exec-1, 1797279991b6ed92,ff9c8fe74c2370b8] INFO c.j.r.i.rest.CargoRoutingController - CargoRoutingController.findOptimalRoute()

**Zipkin:**

Now let assume we have following service calling in order given S1->S2->S3->S4. Suppose S3 has some performance issue and it is very slow. And due to this S2 will be slow and S1 will also be slow. Using zipkin client we can identify which service is slower. Zipkin and sleuth works together. Zipkin can’t work without sleuth. It registers listener to sleuth. Whenever span trace start sleuth will send the span start time and end time. All that information store in zipkin.

1. Run zipkin executable jar.



1. Add below dependency on pom.xml.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

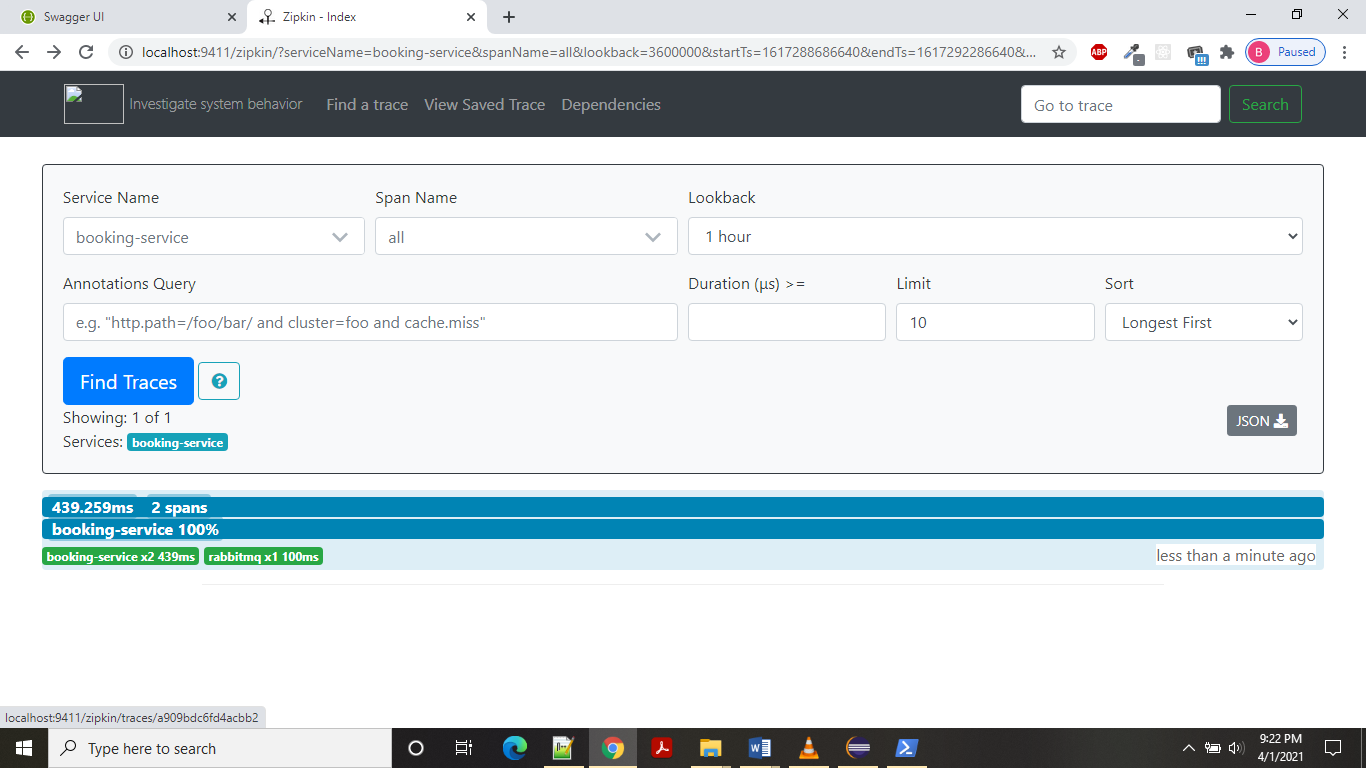
1. Now we can say whenever request come it will send data to zipkin due to this we can say it may impact performance that’s why these calls are asynchronous. Now suppose if we have 1000 request we don’t want to send 1000 request data to zipkin client to identify which service is slow. So, for that reason we can send 10% of request to zipkin server. For this we can configure below property. 1.0 means 100%

sleuth:

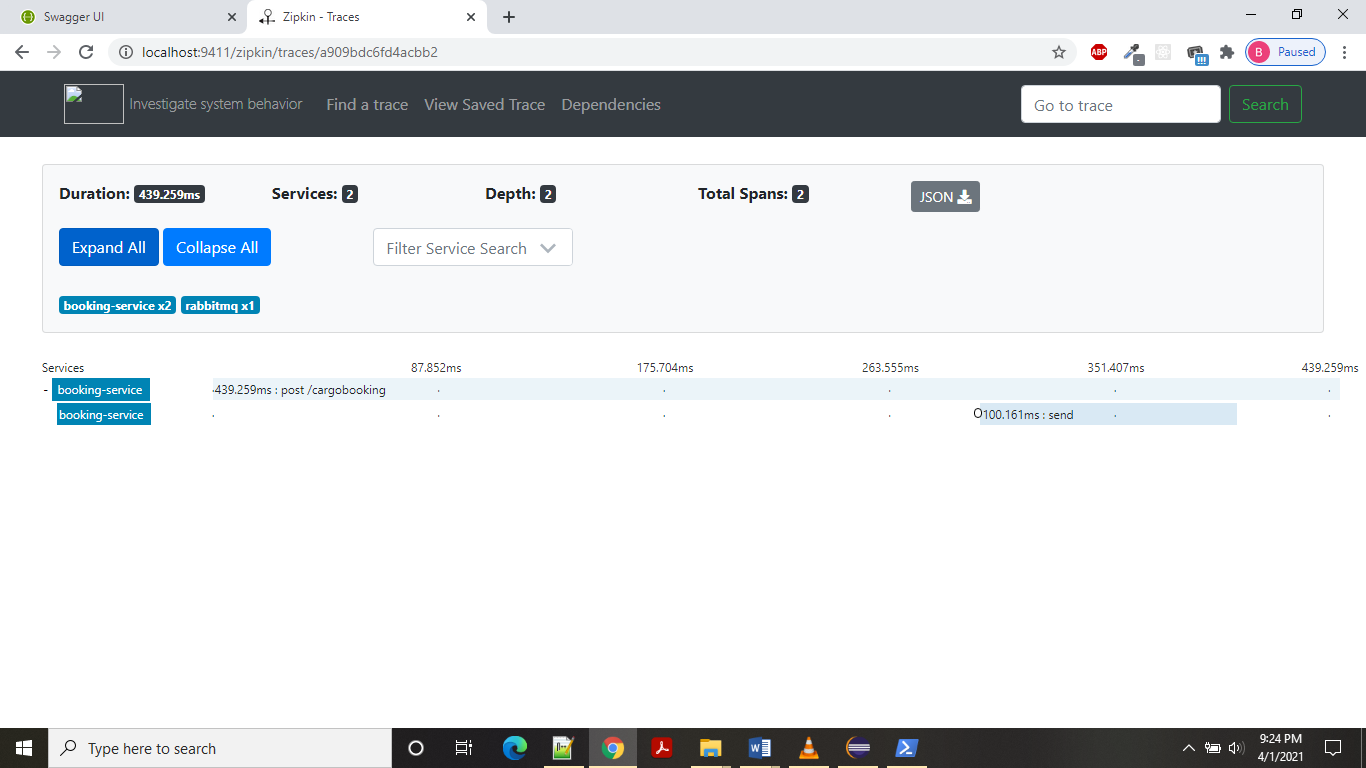
sampler:

probability: 1.0

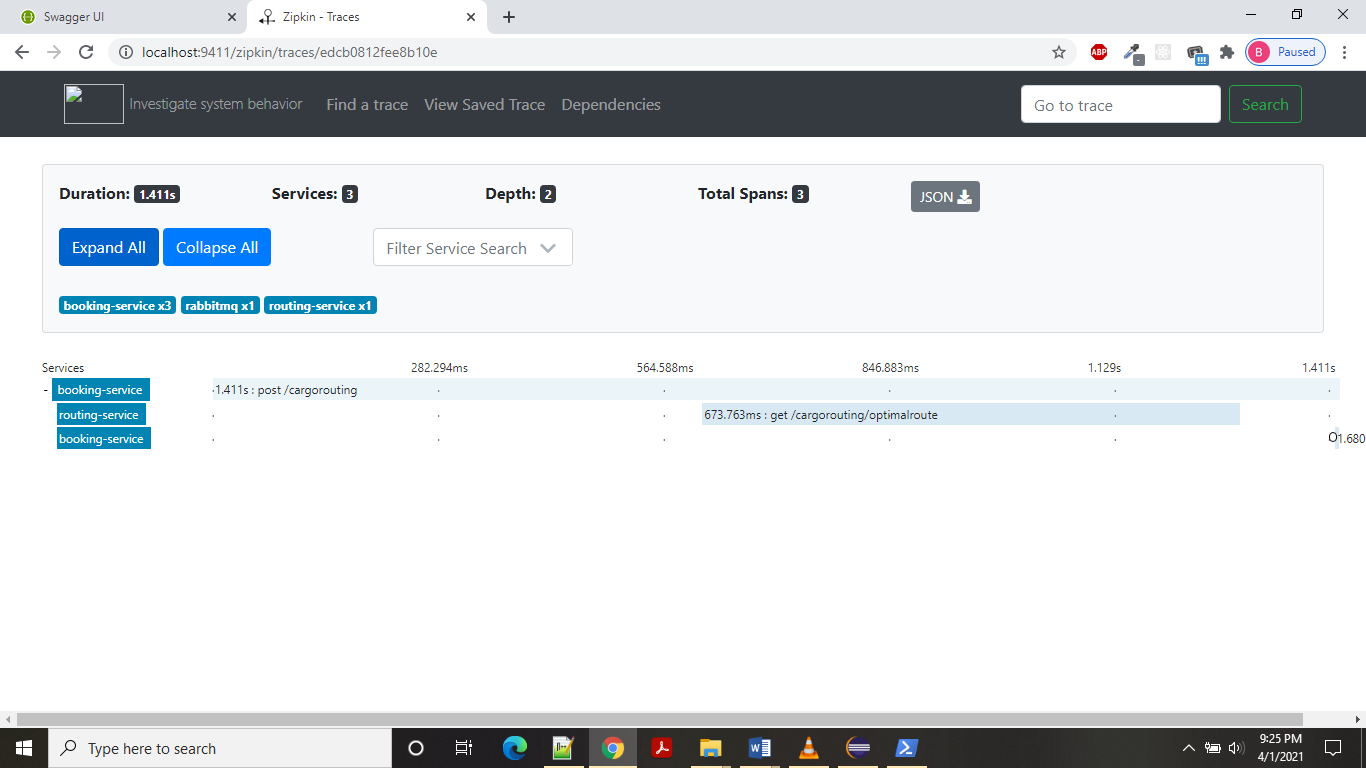
1. After configuring these property restart all the service.
2. Go to zipkin UI but there you can’t see any service so let’s open swagger-ui.html. and create a cargo -booking.



Select service name and click on find traces. Click on blue block and you can see the timing



Now here we can’t see call for routing service. Now let’s call routing rest service using booking id.



So in above screenshot you can see how much time it is actually taking. And based on it you can identify which service is taking more time and based on this you can fix this.