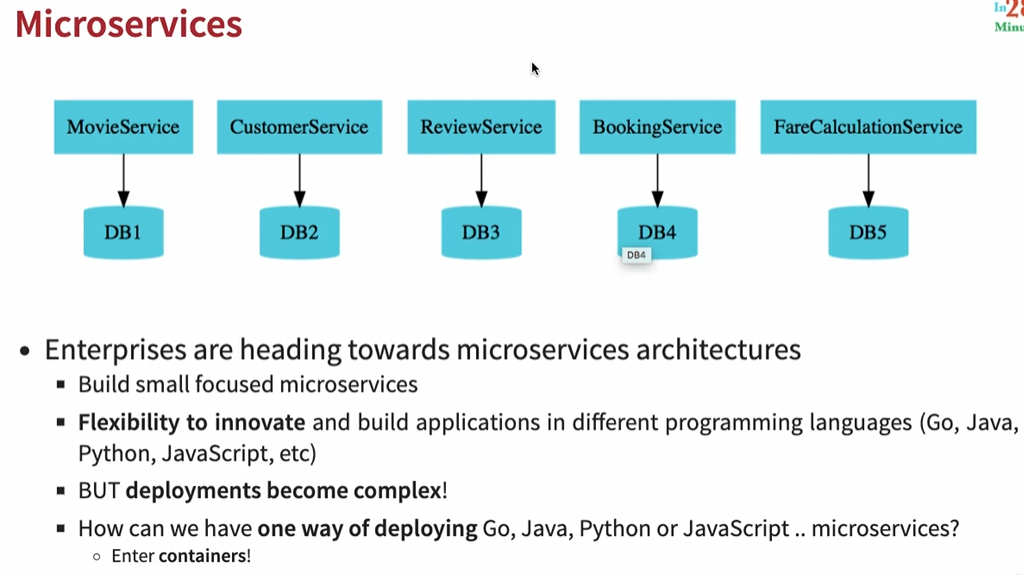
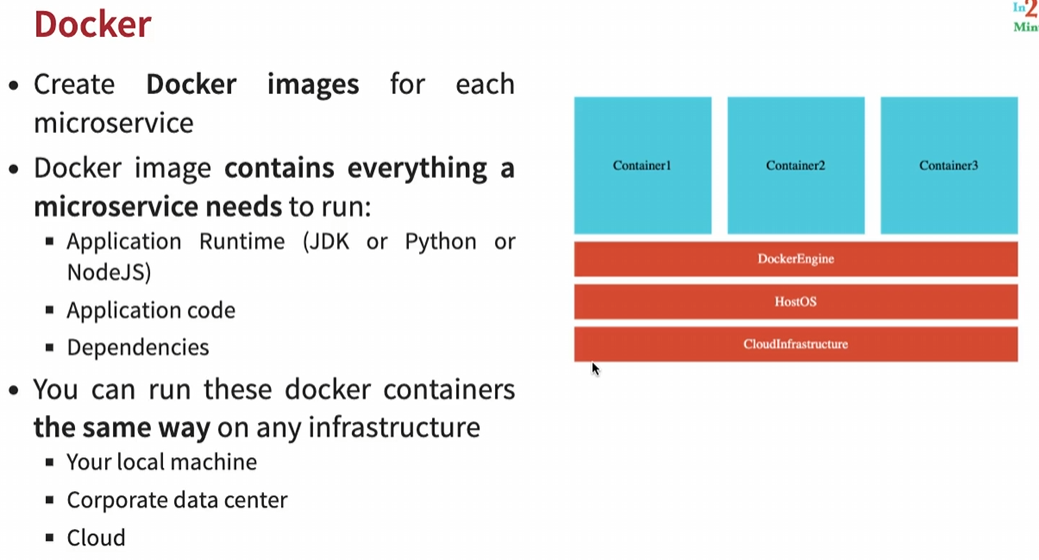
**Microservices Docker**



* The most popular container tool is **Docker.**



**Docker Terminologies**

1. Docker Registry

* A registry contains a lot of repositories with different versions of different/same applications. Private or Public repositories are there.
* The image contains all the things that our application needs in order to run like right software, specific java version or libraries or any other dependency that our application might need.
* Container image is something static so in repository/ local pull, image is a set of bytes & when image is running, it’s called a container.
* So, Image is static version while Container is a running version & for the same image, we can have multiple containers running.
* For executing the API calls inside the image, we need to map container port to our local m/c port using option => **-p** {HostPort}:{ContainerPort}
* Any container that we run is part of bridge network in Docker like internal docker network, nobody will be able to access it unless we specifically expose it onto the host/system where container is running.
* Demo Program to run in docker cli

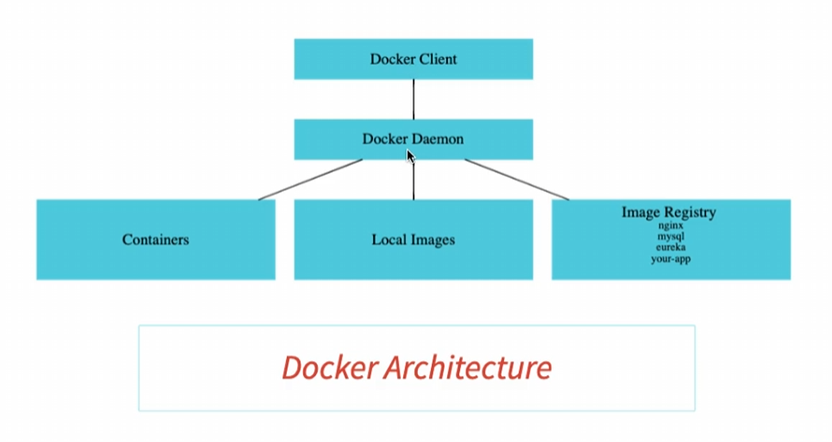
Command**: docker run -p** 5000:5000 in28min/todo-rest-api-h2:1.0.0.RELEASE

API Url: <http://localhost:5000/hello-world>

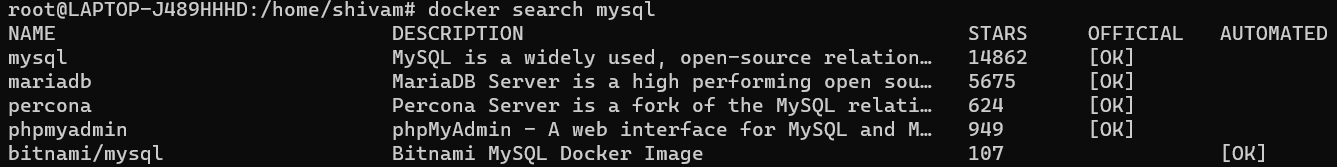
API Url: <http://localhost:5000/hello-world-bean>

* If we want to run our application in background, we can use option => -d => detached mode
* Command: **docker run -p** 5000:5000 -d in28min/todo-rest-api-h2:1.0.0.RELEASE
* Command to see local docker images: **docker images**
* Command to see running container: **docker ps / docker container ls**
* Command to see logs: **docker log** container\_id
* Command to follow logs: **docker log -f** container\_id
* Command to stop container: **docker stop** container\_id / **docker container stop** container\_id

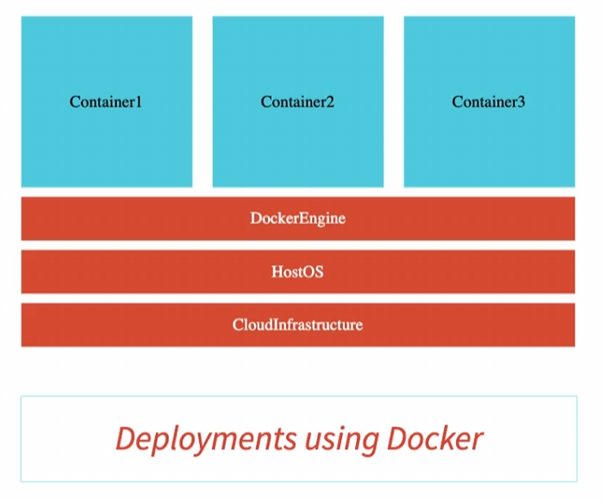
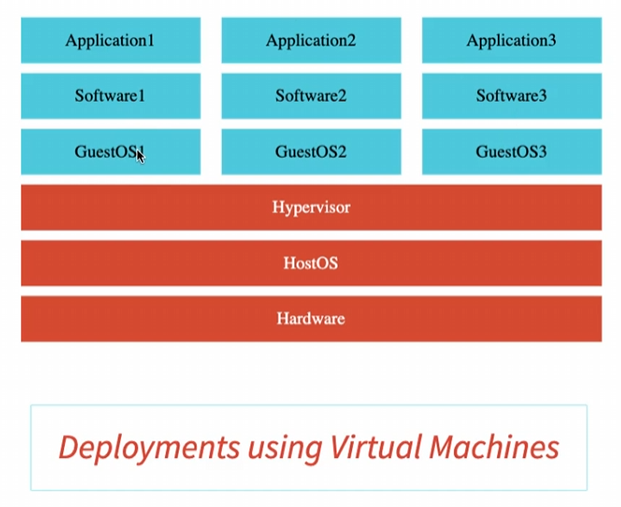
**Docker Architecture**



* **Docker Client:** The place where we’re running the commands in is called a Docker client.
* **Docker Daemon / Docker Engine:** When we type something in Docker client, the command is sent out to something called a Docker Daemon or a Docker engine for execution.
* Docker Daemon is responsible for managing the containers & local images. Also responsible for pulling something from image registry or pushing a locally created image to a image registry.
* We can install docker locally as well as on cloud very easily. Most of the cloud providers actually provide container-based services.
* Docker is relatively light weight with respect to Virtual m/c & therefore it’s very efficient.
* Azure provides a service called Azure Container Service.
* AWS provides a service called Elastic Container Service.
* **Docker Official Images:** They are a curated set of Docker repositories hosted on docker hub i.e., basically Docker has a team which looks at these images & make sure that these images are meeting certain standards, & they publish all these content in the official images.
* So, whenever you’d want to use image for some software, make sure that you’re using an official image.
* Command to check for official images: **docker search** [imageName]



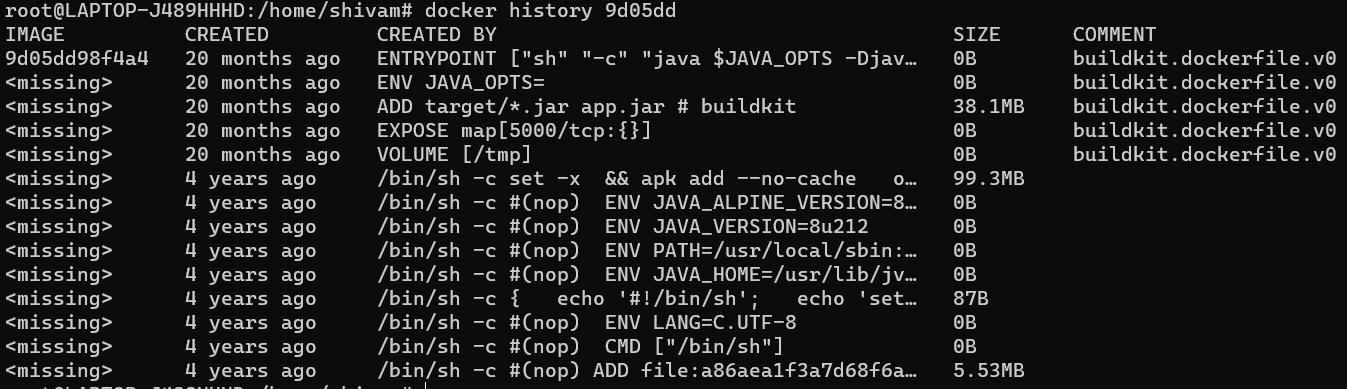
**Deployment using VM Vs Docker**



* Before Docker, Virtual machines were very popular. In virtual m/c we have H/w layer then Host OS layer installed on top of the H/w & we have Hypervisor on top of Host OS to manage our virtual machines (VM).
* Now Each VM had a guest OS, Software & on the top of these will be our application.
* One of the major problems with this virtual m/c architecture, was typically these are heavy weight i.e., we had 2 operating system that makes the whole thing heavy.
* This is where Docker comes in. On the top of host OS, all we need to do is install the Docker engine for that specific Host OS like Win, mac, Linux & Docker would take care of managing these containers. The Docker image contains all that is needed to run a container i.e., all libraries, softwares etc are directly part of these containers.
* Because there is only one Host OS, Docker is relatively light weight & therefore is very efficient.

**Playing with Docker images**

* We can have multiple tag to an image
* Command: **docker tag** [repository:oldTag] [repository:newTag]
* E.g., docker tag in28min/todo-rest-api-h2:1.0.0.RELEASE in28min/todo-rest-api-h2:latest
* **Imp**. Latest tag for any repository doesn’t always specify that it’s the latest release so, it’s better to go to repository, check the tags & then pull the image.
* We can use the command => **docker pull** [imageName] => to pull image from the repository/docker registry but it won’t run the image. For e.g., => **docker pull** mysql
* Command => **docker run** … => will check if image is available local or not, if not available locally, it will pull it & creates a container.
* Command => **docker history** [imageName] => will show the steps involved in creating that specific image



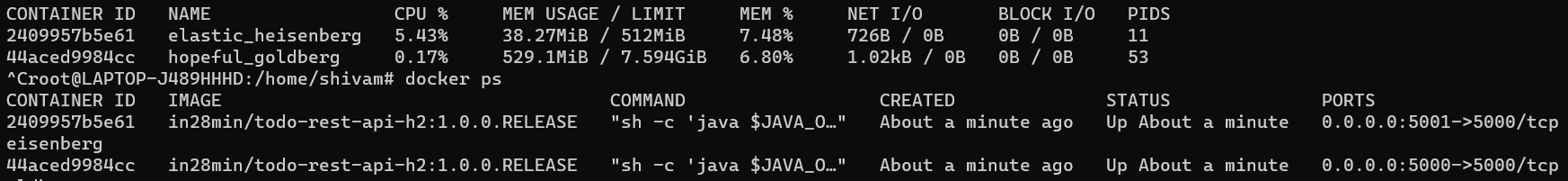
* Command => **docker inspect** [imageName] => inspect a particular image.
* Command => **docker remove** [imageName] => removes the image locally not from repository.

**Playing with Docker Container**

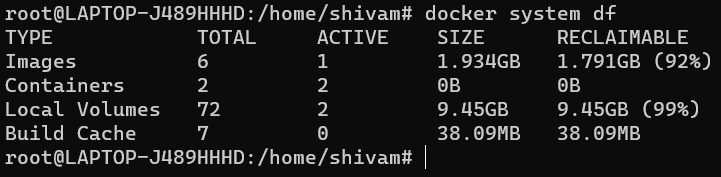
* Command => **docker container run** -p 5000:5000 -d in28min/todo-rest-api-h2:1.0.0.RELEASE
* Command => **docker container pause** [container\_id]
* Command => **docker container unpause** [container\_id]
* Command => **docker container** **stats**
* Command => **docker container prune** => will delete all the exited containers but not running containers.
* Command => **docker container stop** [container\_id] => will shut down the running container gracefully.
* Command => **docker container kill** [container\_id] => will kill the running container forcefully.
* We can add restart policy to docker run command. If in case docker daemon stops or restart, still our container will restart.
* Command => docker run -p 5000:5000 -d -restart=always [repository:release]

**Playing with Docker Commands**

* Command => **docker events** => it helps us to see what heap events are happening with docker.
* Command => **docker top** [container\_id] => give all the processes which are running in that container.
* Command => **docker stats** => show all the stats regarding the containers which are running.
* We can assign memory, CPU to a specific container configuration based on our requirement
* Command => **docker run** -p 5001:5000 -m 512m --cpu-quota 5000 -d [repo:release]



* Command => **docker system df** => will show all the different things that our docker daemon manages like Images, containers.



**Observability & OpenTelemetry**

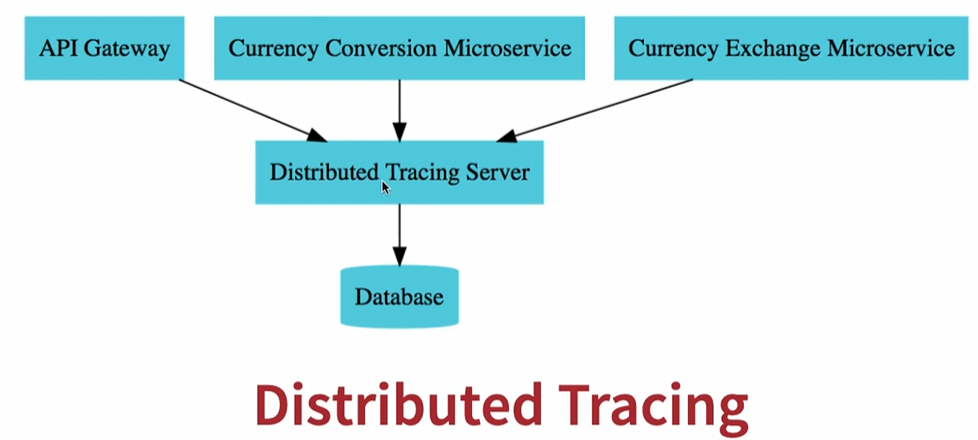
* **Monitoring Vs Observability**
* Monitoring is reactive. Monitoring is all about looking at metrics, logs, traces. We’re looking at what has happened.
* Monitoring is a subset of Observability
* Observability is proactive. Observability focuses on, how well do we understand what’s happening in a system. It focuses on getting intelligence from the data that helps us to detect problems faster.
* Step 1: Gather data: metrics, logs or traces (Monitoring)
* Step 2: Get Intelligence: AI/Ops & anomaly detection
* OpenTelemetry
* Collection of tools, APIs & SDKs to instrument, generate, collect, & export telemetry data (metrics, logs, & traces)
* Instead of having different standards for different things, we can have just one standard for all the things related to gathering data. This is where OpenTelemetry comes into picture.
* Today, almost every cloud platform AWS, Azure, Google Cloud provides support for telemetry in one form or another.

**Microservice deployment using Docker**

**Distributed Tracing**



* Microservices typically have complex call chains & because of that, Tracing Request across microservices, finding where the problem is & Debugging Problem is a tough task to do.
* This is where we go for a feature called **Distributed Tracing.**

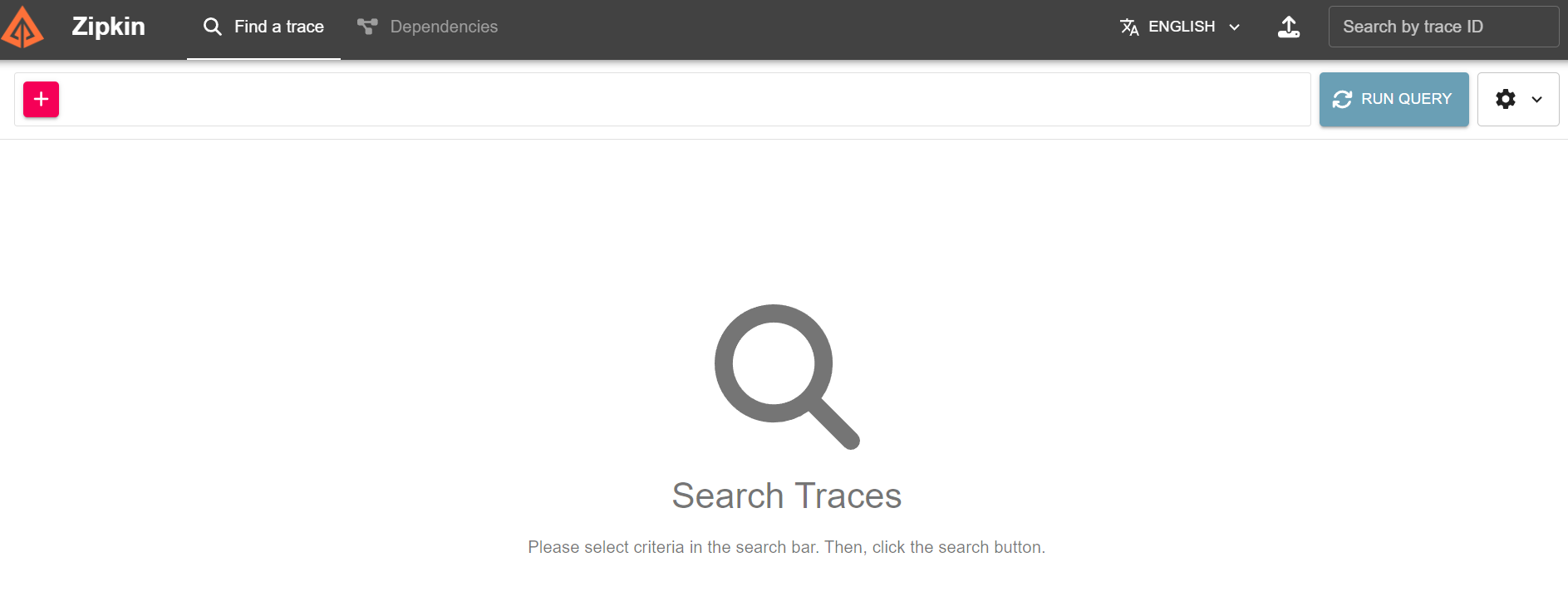


* All these microservices involved in complex call chains would send all the information out to a single distributed tracing server, & this server would store everything to a database (in memory or real).
* This tracing server would provide an interface which would allow us to trace the request across multiple microservices.

**Distributed Tracing Server – Zipkin using Docker**



* Here Instead of creating a java project for Zipkin, we’re launching it as a docker container.
* Command: **docker run** -p 9411:9411 **openzipkin/zipkin**
* Url to access Zipkin server: <http://localhost:9411>



**Scenario where we can use Distributed Tracing**

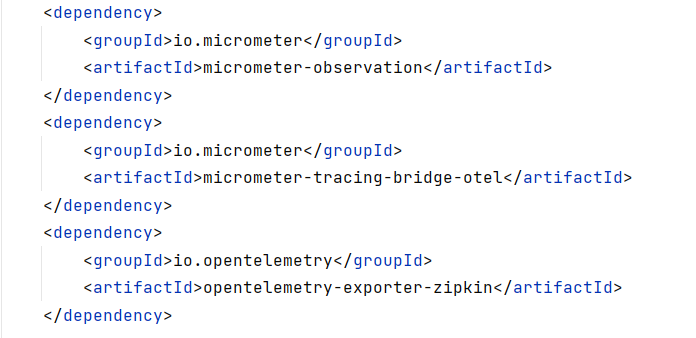
* A request is flowing through multiple microservices & it’s talking a lot of time. We want to find out how much time, the request is spending in each microservice & we want to find out which microservice is consuming the most amount of time.
* This is where Distributed tracing is really, really useful.
* The way distributed tracing would work is that each of the microservices would send the tracing information out to the distributed server & the distributed tracing server would have all that information stored in a database & would provide a UI where we can query & find information about the requests which are executed.

**Distributed Tracing Configuration for Spring Boot 2.4.x & 3.x.x version**

* There is significant change in configuration depending on spring boot versions.
* We can refer this url: <https://github.com/in28minutes/spring-microservices-v3/blob/main/v3-upgrade.md>
* In Spring Boot 2.4.x, the recommended approach was using Spring Cloud Sleuth. This would send the information to tracing library (Brave) & from brave, we would send the information out to Zipkin.



* In Spring Boot 3.x.x, the recommendation is to use **Micrometer**. **Sleuth** can only handle traces but Micrometer can handle observations: basically, metrics and logs & traces as well.
* Micrometer is a vendor – neutral application observability façade. It can instrument our JVM-based application code without vendor lock-in.
* Instead of Brave as tracing library or Bridge, we will use **OpenTelemetry** (an open standard for metrics, logs, & tracing) as the bridge.



* In Spring Boot 3, we can use Brave as the bridge but it would only support tracing not metrics & logs.
* So, Basically from Spring Boot 2 to 3, we’re moving from trace-specific configuration to open standards.

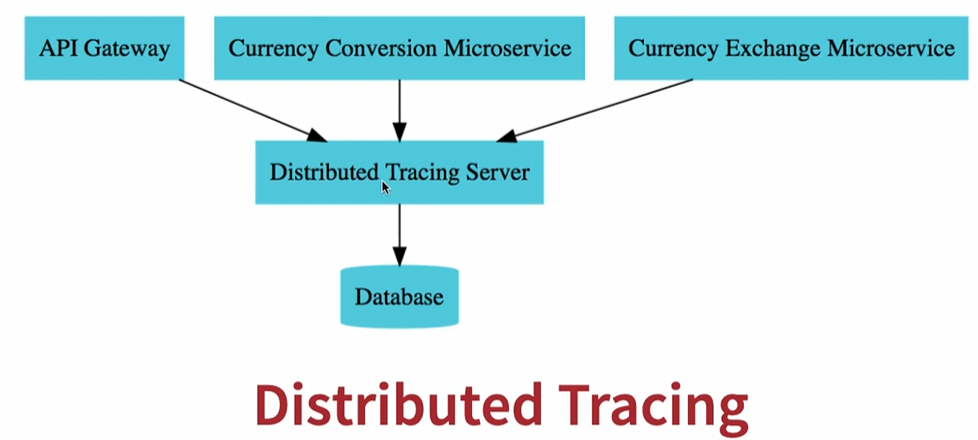
**Sampling**

* If we trace all the requests which are coming to all the microservices, then there will be a big performance impact & that’s why we can configure how much percentage of the request we’d want to sample.



* Here probability = 1.0 means we would want to trace every request. Probability = 0.05 means we would want to trace only 5% of the requests.

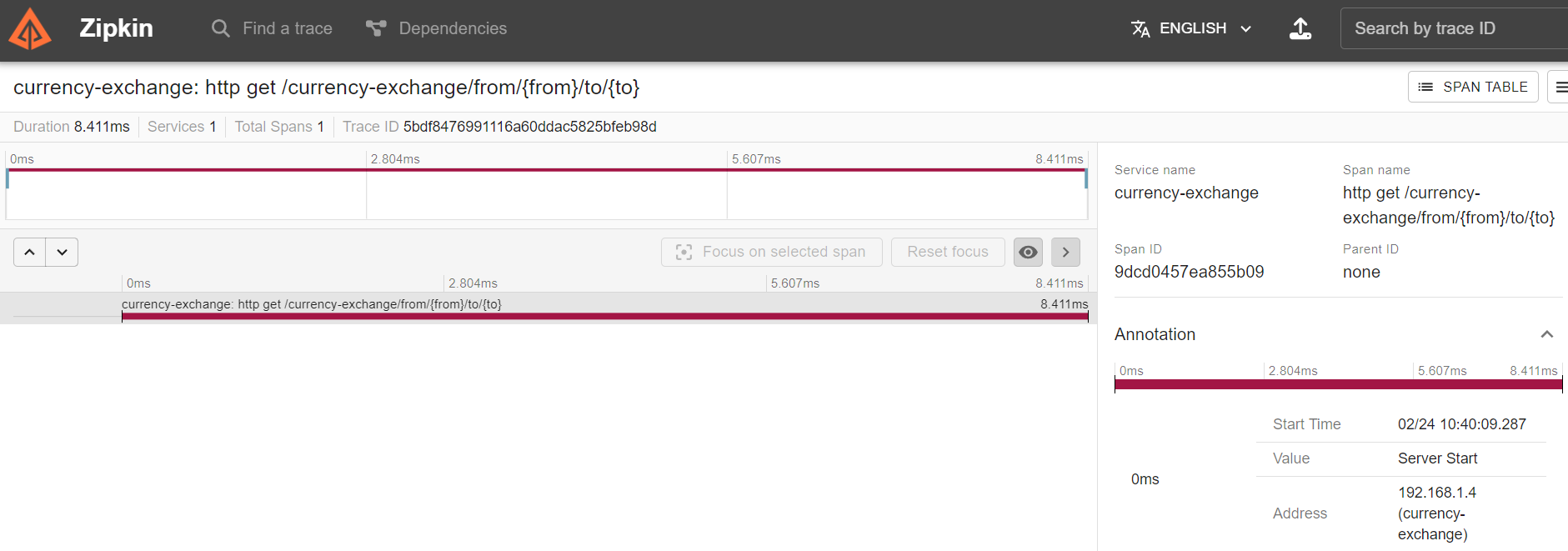
**Adding Tracing configuration to Microservices**



**1. currency-exchange-microservice**

* Add above dependencies to pom.xml
* In order to trace a request across multiple microservices, the request needs to have an ID assigned to it. That’s what Micrometer is doing.
* Micrometer would assign an ID to the specific request for tracing. The ID will be retained across multiple microservices.
* i.e., when the request goes from currency-conversion-ms to currency-exchange-ms, the ID will be retained.
* Becoz of provided logging pattern, we can see our log as below (traceId = 5bd.., spanId = 9dcd..)

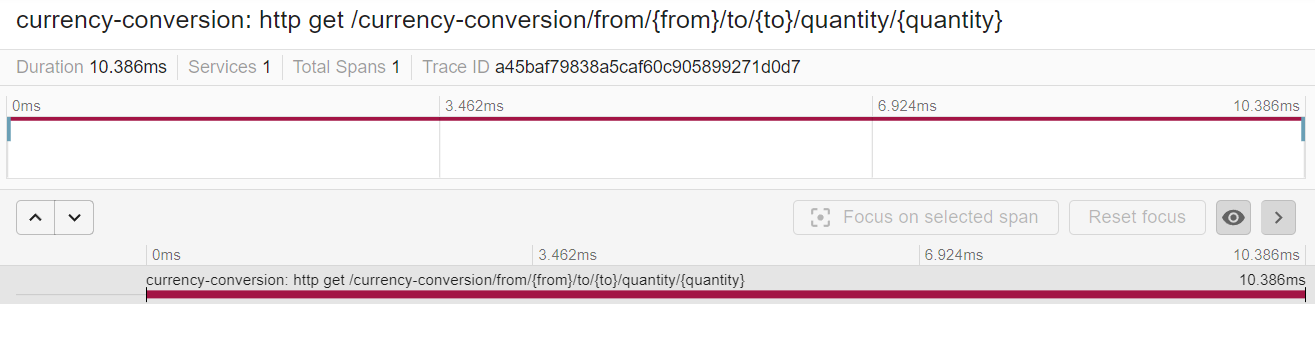




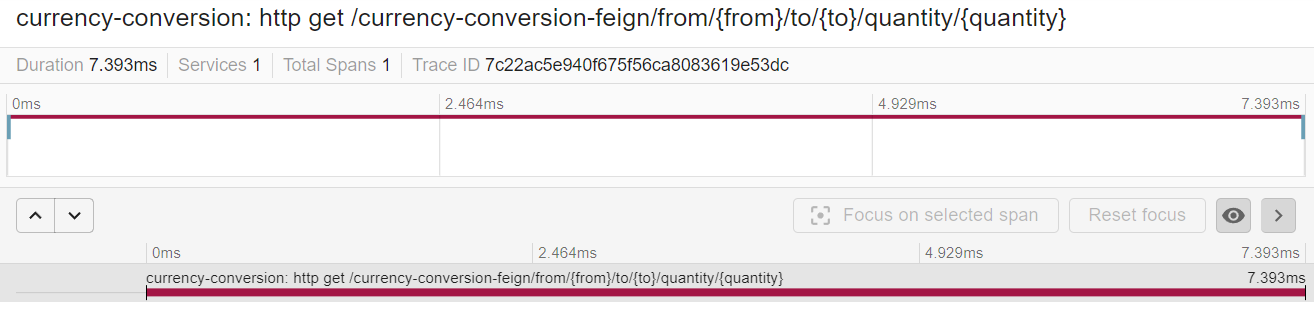
**2. currency-conversion-microservice & API Gateway**

* We can integrate same dependencies to currency-conversion ms & API Gateway. Now let’s check zipkin

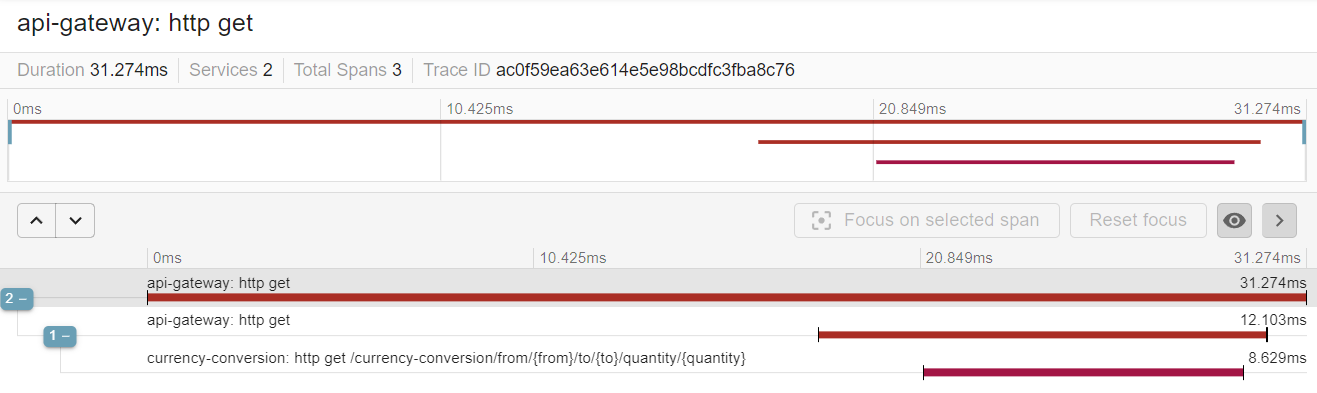
**Url**: <http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/100>



**Url**: <http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/100>



**Url**: <http://localhost:8765/CURRENCY-CONVERSION/currency-conversion/from/USD/to/INR/quantity/10>



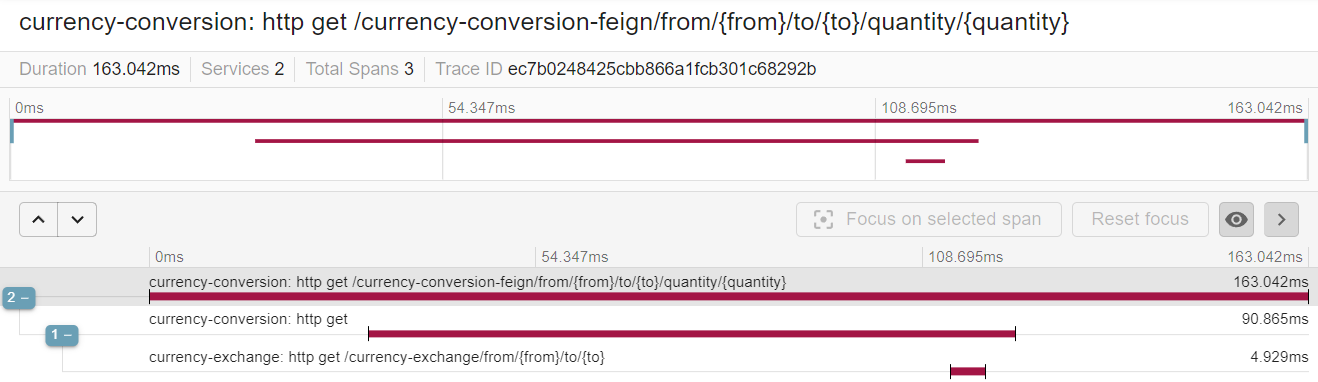
**Problem:** Here we can currency-conversion-ms is calling currency-exchange-ms using RestTemplate & Feign Client & we can see traces from API gateway to currency-conversion-ms, but the call to currency-exchange-ms, we’re unable to trace.

* **Solution for Feign Client**: In Spring Boot 3, there is an additional configuration we need to add. Add dependency for **Feign Client**.

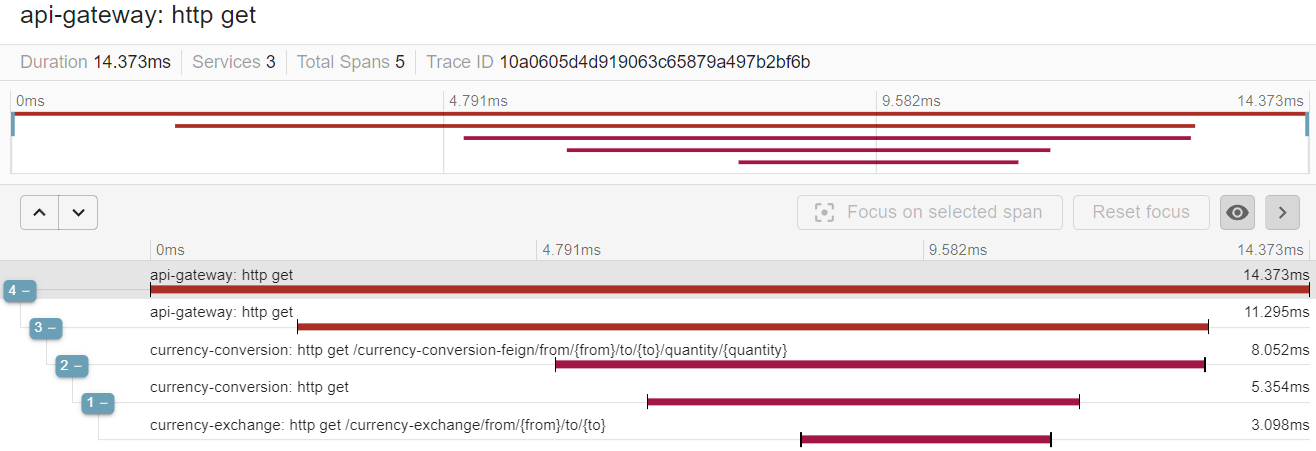


**Url**: <http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/100>

* We can see currency-exchange-ms trace now



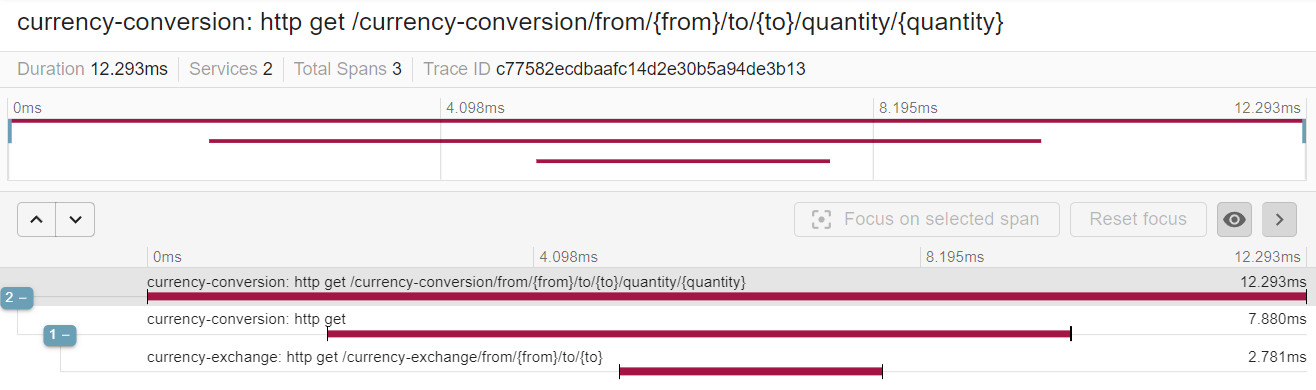
**Url**: <http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/100>



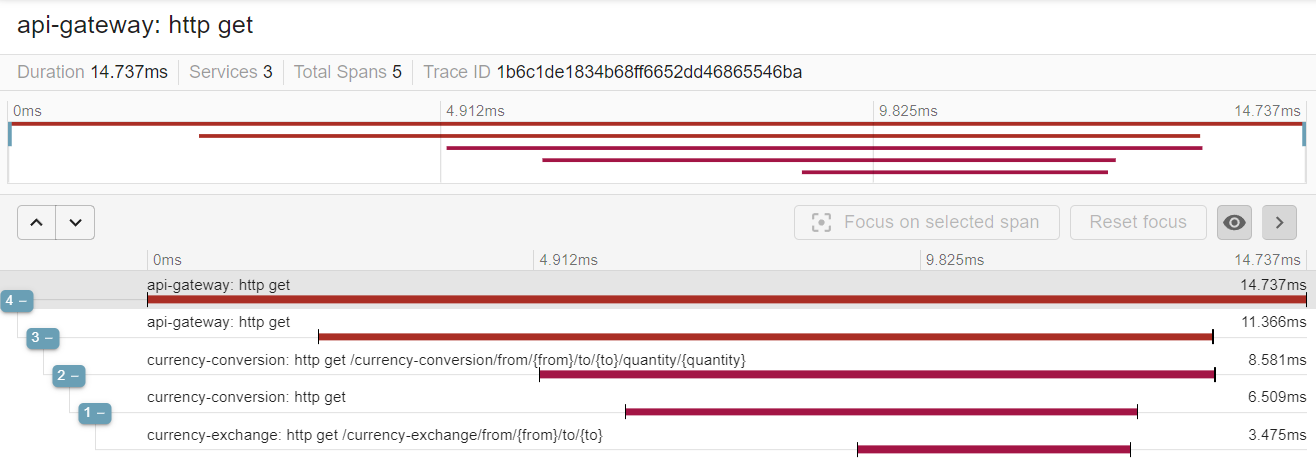
* **Solution for RestTemplate:** In Spring Boot 3, we need to create an explicit RestTemplate bean & integrate RestTemplate with Micrometer.



**Url:** <http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10>



**Url:** <http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10>



**Docker Deployment**

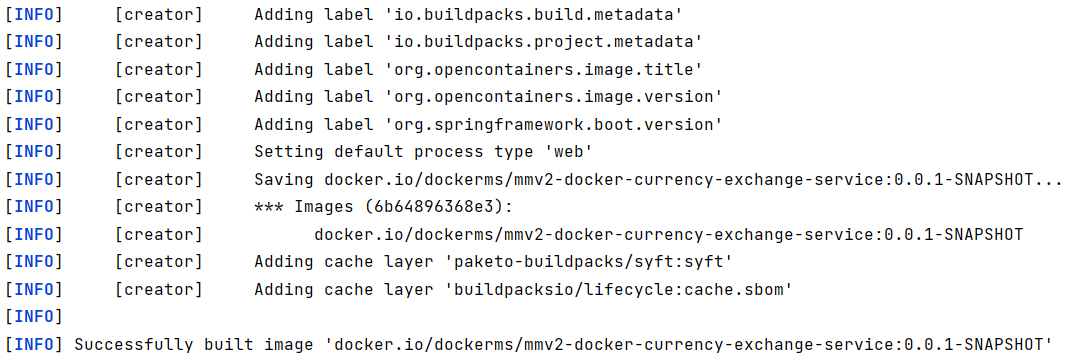
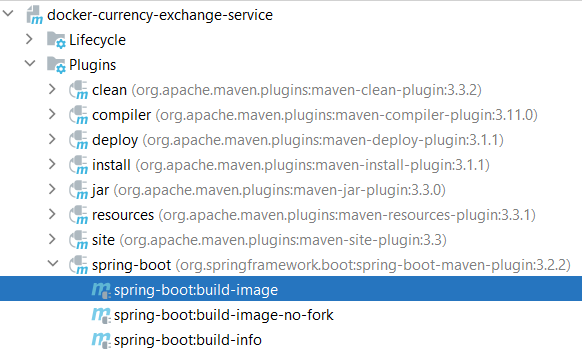
* Before Spring Boot version 2.3, there was a lot of process involved in creating a Docker container for Spring Boot application or Java applications.
* But from Spring Boot version 2.3, it’s very easy to create Docker containers.

**Step 1:** All we need to do is to use maven-plugin (already in pom.xml). Add below configuration:



Here, **dockerms** is the docker id & **mmv2-${project.artifactId}** is the image name.

**Step 2:** Now build the image using maven build & goal as **spring-boot:build-image**



**Step 3:** Now the docker image is built. We can check the image using below command & run

* Command: **docker images**
* Command: **docker run** -p 8000:8000 [imageId] or [builtImageName]
* **builtImageName**: dockerms/mmv2-docker-currency-exchange-service:0.0.1-SNAPSHOT

We can follow above steps for all the microservices & execute the urls to test.

**Docker Compose**

* Building each microservice & separately deploying each microservices is not going to be a easy thing & that’s the reason why we go for something called Docker Compose.
* Docker compose is a tool for defining & running multi container docker applications.
* We can simply configure a YAML file, & with a single command we can launch up all these services which are defined inside the YAML file.

**docker-compose.yaml** (**Command**: docker-compose up)

|  |
| --- |
| version: '3.7'  services:  docker-naming-server:  image: dockerms/mmv2-docker-naming-server:0.0.1-SNAPSHOT  mem\_limit: 700m  ports:  - "8761:8761"  networks:  - currency-network   zipkin-server:  image: openzipkin/zipkin:latest  mem\_limit: 300m  ports:  - "9411:9411"  networks:  - currency-network   docker-currency-exchange:  image: dockerms/mmv2-docker-currency-exchange-service:0.0.1-SNAPSHOT  mem\_limit: 700m  ports:  - "8000:8000"  networks:  - currency-network  depends\_on:  - docker-naming-server  environment:  EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://docker-naming-server:8761/eureka  MANAGEMENT.ZIPKIN.TRACING.ENDPOINT: http://zipkin-server:9411/api/v2/spans   docker-currency-conversion:  image: dockerms/mmv2-docker-currency-conversion-service:0.0.1-SNAPSHOT  mem\_limit: 700m  ports:  - "8100:8100"  networks:  - currency-network  depends\_on:  - docker-naming-server  environment:  EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://docker-naming-server:8761/eureka  MANAGEMENT.ZIPKIN.TRACING.ENDPOINT: http://zipkin-server:9411/api/v2/spans   docker-api-gateway:  image: dockerms/mmv2-docker-api-gateway:0.0.1-SNAPSHOT  mem\_limit: 700m  ports:  - "8765:8765"  networks:  - currency-network  depends\_on:  - docker-naming-server  environment:  EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://docker-naming-server:8761/eureka  MANAGEMENT.ZIPKIN.TRACING.ENDPOINT: http://zipkin-server:9411/api/v2/spans  networks:  currency-network: |