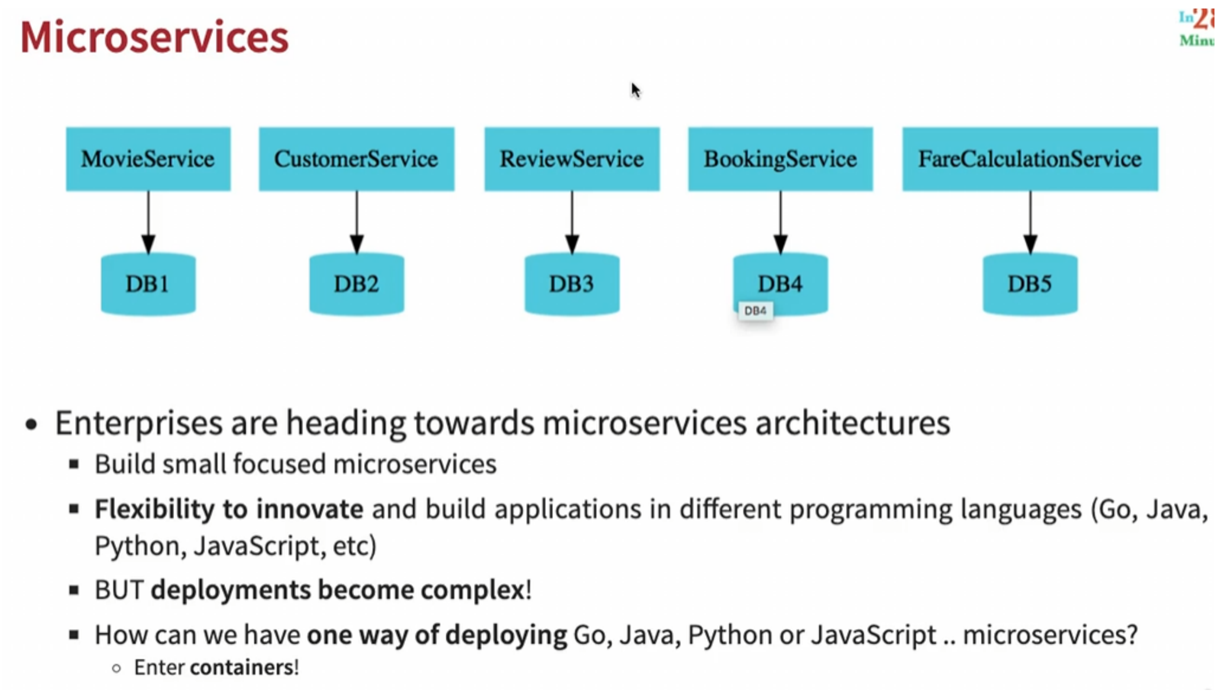
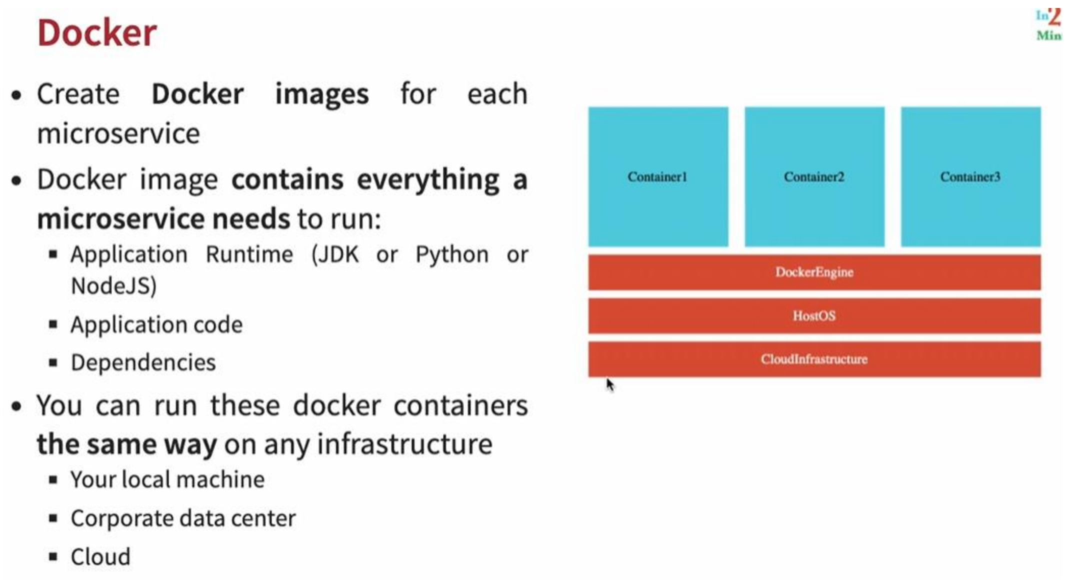
Microservices Docker



Microservices

- Enterprises are heading towards microservices architectures
  - Build small focused microservices
  - **Flexibility to innovate** and build applications in different programming languages (Go, Java, Python, JavaScript, etc)
  - BUT **deployments become complex**!
  - How can we have **one way of deploying** Go, Java, Python or JavaScript .. microservices?
    - Enter **containers**!

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



Docker

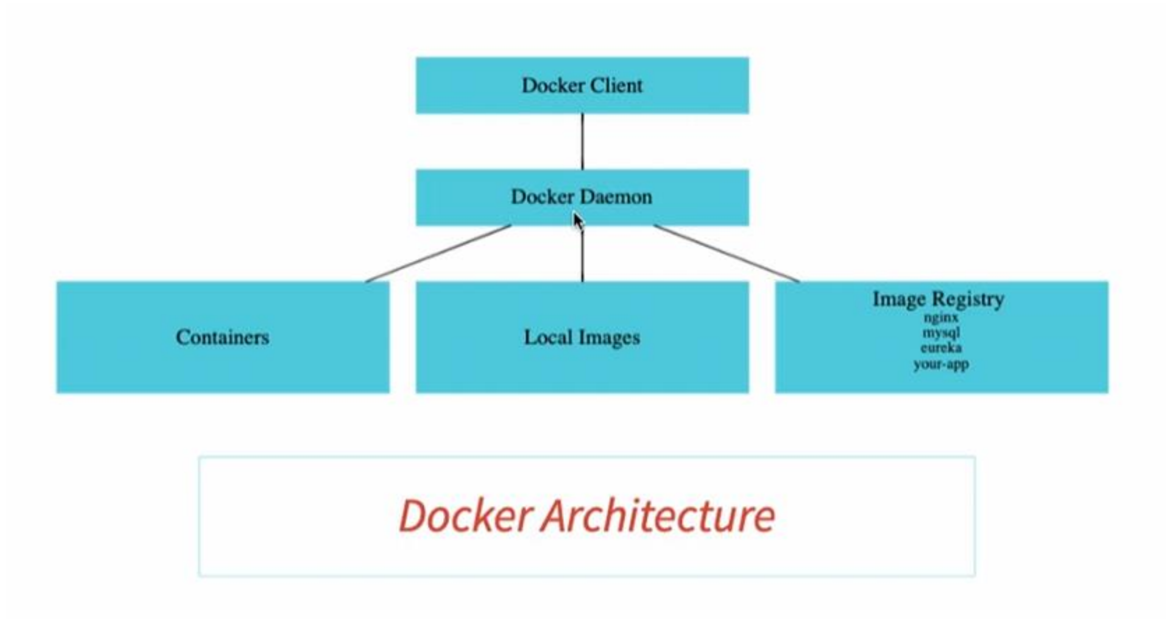
- Create **Docker images** for each microservice
- Docker image **contains everything a microservice needs** to run:
  - Application Runtime (JDK or Python or NodeJS)
  - Application code
  - Dependencies
- You can run these docker containers **the same way** on any infrastructure
  - Your local machine
  - Corporate data center
  - Cloud

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

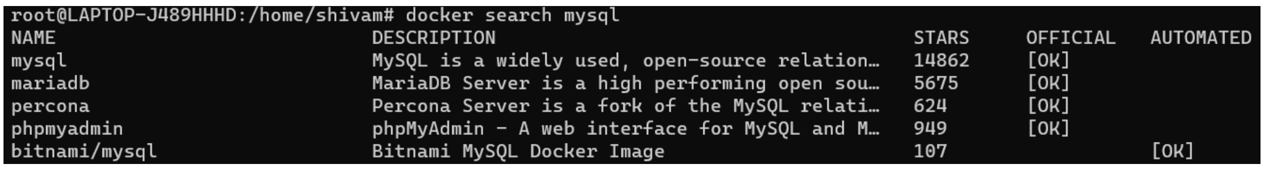
Docker Architecture



- **Docker Client:** The place where we're running the commands in is called a Docker client.

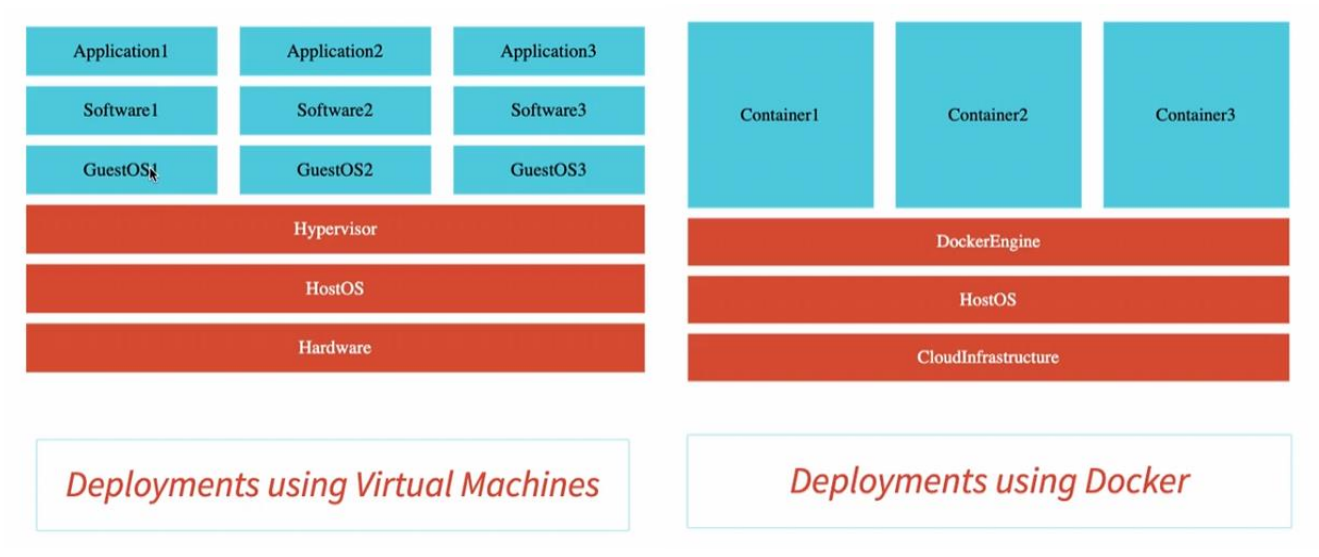
- **Docker Daemon / Docker Engine:** When we type something in Docker client, the command is sent out to something called a Docker Daemon or a Docker engine for execution.
- Docker Daemon is responsible for managing the containers & local images. Also responsible for pulling something from image registry 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]



```
root@LAPTOP-J489HHHD:/home/shivam# docker search mysql
NAME                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
mysql               MySQL is a widely used, open-source relation…   14862     [OK]
mariadb             MariaDB Server is a high performing open sou…   5675      [OK]
percona             Percona Server is a fork of the MySQL relati…   624       [OK]
phpmyadmin          phpMyAdmin - A web interface for MySQL and M…   949       [OK]
bitnami/mysql       Bitnami MySQL Docker Image                      107                  [OK]
```

Deployment using VM Vs Docker



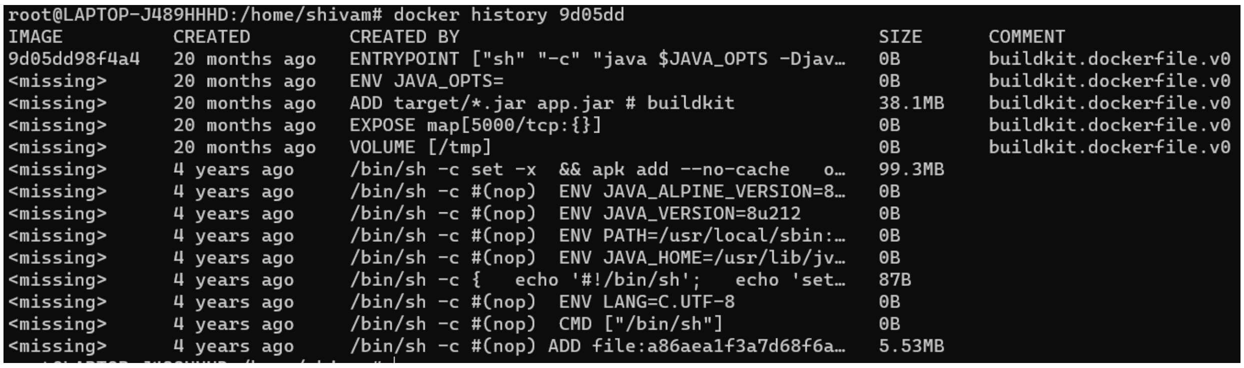
- Before Docker, Virtual machines were very popular. In virtual m/c we have H/w layer then Host OS layer installed on top of the H/w & we have Hypervisor on top of Host OS to manage our virtual machines (VM).
- Now Each VM had a guest OS, Software & on the top of these will be our application.
- One 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



```
root@LAPTOP-J489HHHD:/home/shivam# docker history 9d05dd
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
9d05dd98f4a4   20 months ago   ENTRYPOINT ["sh" "-c" "java $JAVA_OPTS -Djav…   0B        buildkit.dockerfile.v0
<missing>      20 months ago   ENV JAVA_OPTS=                                  0B        buildkit.dockerfile.v0
<missing>      20 months ago   ADD target/*.jar app.jar # buildkit             38.1MB    buildkit.dockerfile.v0
<missing>      20 months ago   EXPOSE map[5000/tcp:{}]                         0B        buildkit.dockerfile.v0
<missing>      20 months ago   VOLUME [/tmp]                                   0B        buildkit.dockerfile.v0
<missing>      4 years ago     /bin/sh -c set -x  && apk add --no-cache   o…   99.3MB
<missing>      4 years ago     /bin/sh -c #(nop)  ENV JAVA_ALPINE_VERSION=8…   0B
<missing>      4 years ago     /bin/sh -c #(nop)  ENV JAVA_VERSION=8u212       0B
<missing>      4 years ago     /bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:…   0B
<missing>      4 years ago     /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/lib/jv…   0B
<missing>      4 years ago     /bin/sh -c {   echo '#!/bin/sh';   echo 'set…   87B
<missing>      4 years ago     /bin/sh -c #(nop)  ENV LANG=C.UTF-8             0B
<missing>      4 years ago     /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      4 years ago     /bin/sh -c #(nop) ADD file:a86aea1f3a7d68f6a…   5.53MB
```

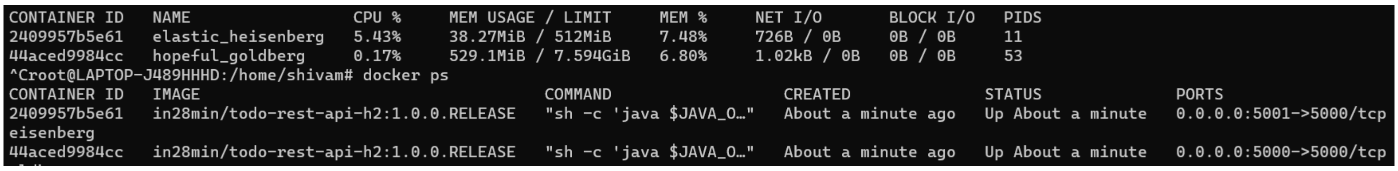
- Command => **docker inspect** [imageName] => inspect a particular image.
- Command => **docker remove** [imageName] => removes the image locally not from repository.

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]

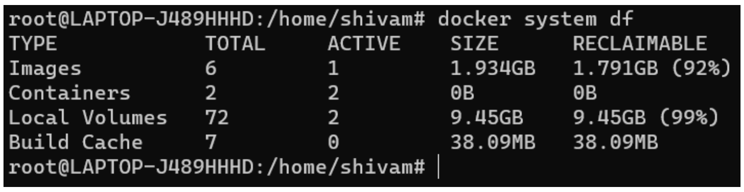
Playing with Docker Commands

- 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]



```
CONTAINER ID   NAME                 CPU %     MEM USAGE / LIMIT     MEM %     NET I/O       BLOCK I/O   PIDS
2409957b5e61   elastic_heisenberg   5.43%     38.27MiB / 512MiB     7.48%     726B / 0B     0B / 0B     11
44aced9984cc   hopeful_goldberg     0.17%     529.1MiB / 7.594GiB   6.80%     1.02kB / 0B   0B / 0B     53
^Croot@LAPTOP-J489HHHD:/home/shivam# docker ps
CONTAINER ID   IMAGE                                  COMMAND                  CREATED              STATUS              PORTS
2409957b5e61   in28min/todo-rest-api-h2:1.0.0.RELEASE   "sh -c 'java $JAVA_O…"   About a minute ago   Up About a minute   0.0.0.0:5001->5000/tcp
eisenberg
44aced9984cc   in28min/todo-rest-api-h2:1.0.0.RELEASE   "sh -c 'java $JAVA_O…"   About a minute ago   Up About a minute   0.0.0.0:5000->5000/tcp
```

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



```
root@LAPTOP-J489HHHD:/home/shivam# docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          6         1         1.934GB   1.791GB (92%)
Containers      2         2         0B        0B
Local Volumes   72        2         9.45GB    9.45GB (99%)
Build Cache     7         0         38.09MB   38.09MB
root@LAPTOP-J489HHHD:/home/shivam#
```

Observability & OpenTelemetry

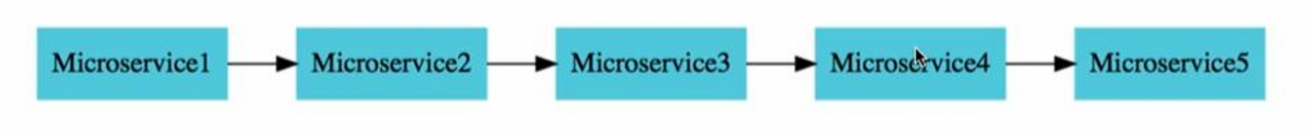
- **Monitoring Vs Observability**
  - Monitoring is reactive. Monitoring is all about looking at metrics, logs, traces. We're looking at what has happened.
    - Monitoring is a subset of Observability

  - 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

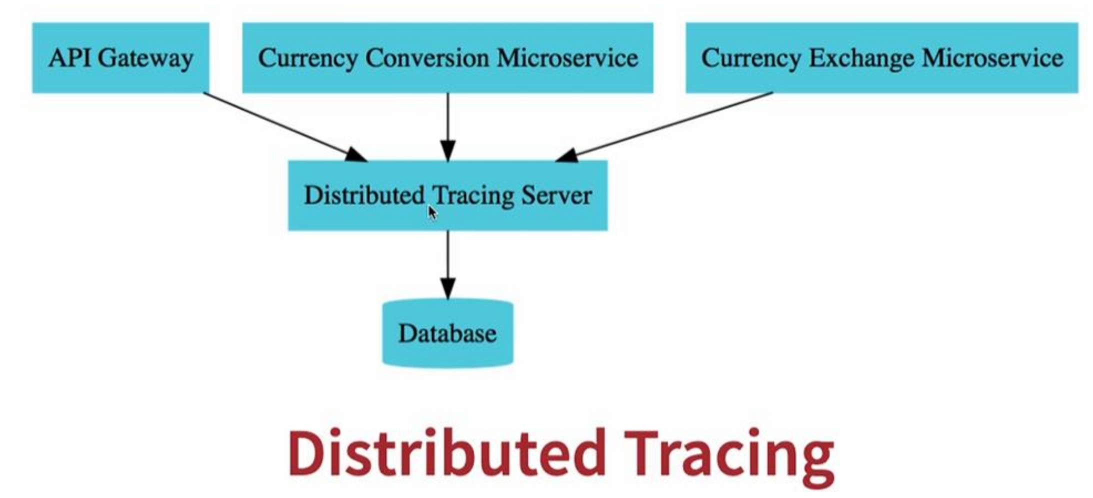
- OpenTelemetry
  - Collection of tools, APIs & SDKs to instrument, generate, collect, & export telemetry data (metrics, logs, & traces)
  - Instead of having different standards for different things, we can have just one standard for all the things related to gathering data. This is where OpenTelemetry 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

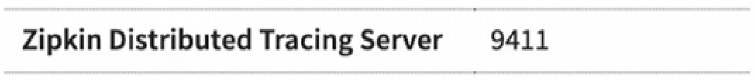


- Microservices typically have complex call chains & because of that, Tracing Request across microservices, finding where the problem is & Debugging Problem is a tough task to do.
- This is where we go for a feature called **Distributed Tracing.**


API Gateway
Currency Conversion Microservice
Currency Exchange Microservice
Distributed Tracing Server
Database
Distributed Tracing

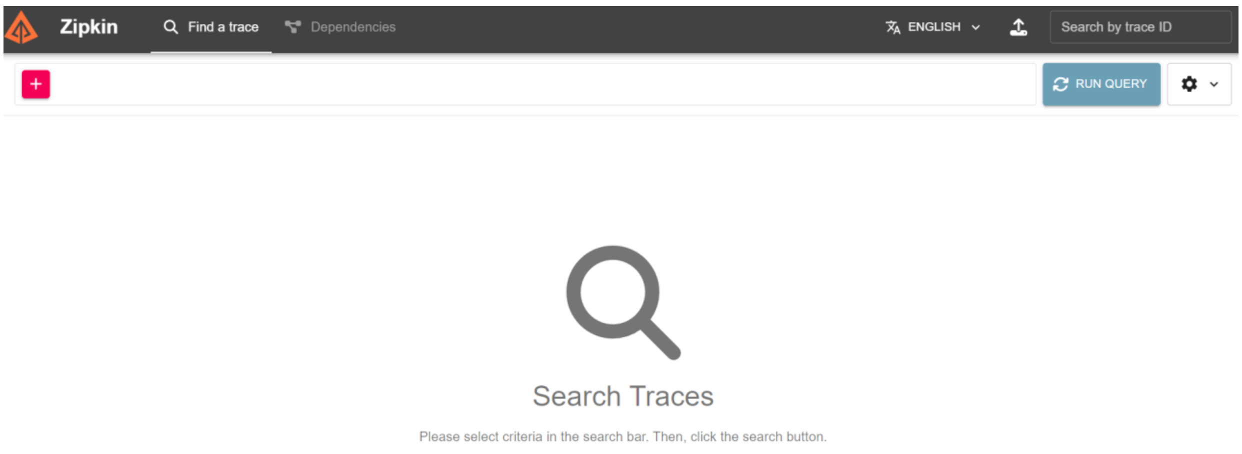
- All these microservices involved in complex call chains would send all the information out to a single distributed tracing server, & this server would store everything to a database (in memory or real).
- This tracing server would provide an interface which would allow us to trace the request across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

Distributed Tracing Server – Zipkin using Docker



Zipkin Distributed Tracing Server	9411

- 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



Search Traces

Please select criteria in the search bar. Then, click the search button.

Scenario where we can use Distributed Tracing

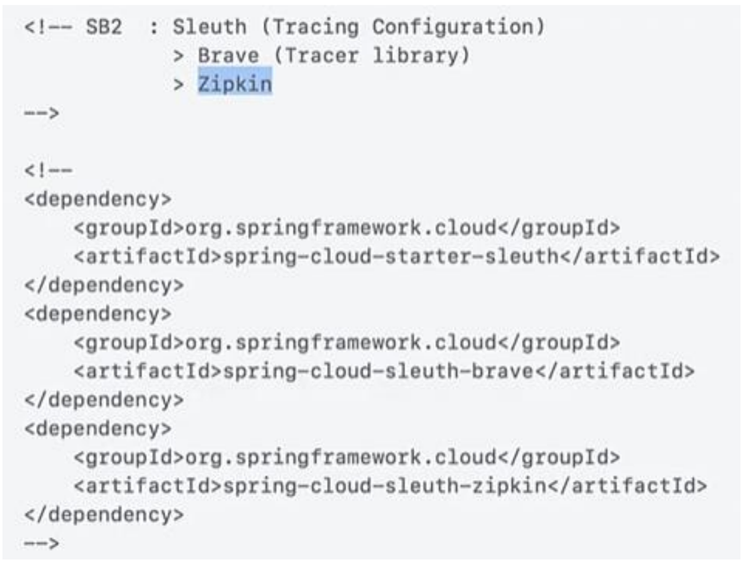
- A request is flowing through multiple microservices & it's talking a lot of time. We want to find out how much time, the request is spending in each microservice & we want to find out which microservice is consuming the most amount of time.

- 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

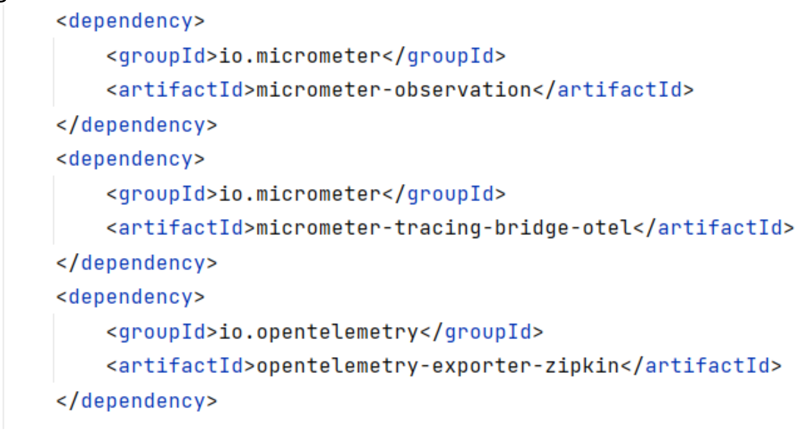
- There is significant change in configuration depending on spring boot versions.
- We can refer this url: https://github.com/in28minutes/spring-microservices-v3/blob/main/v3-upgrade.md

- 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.



```
<!-- SB2  : Sleuth (Tracing Configuration)
            > Brave (Tracer library)
            > Zipkin
-->

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

- 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.

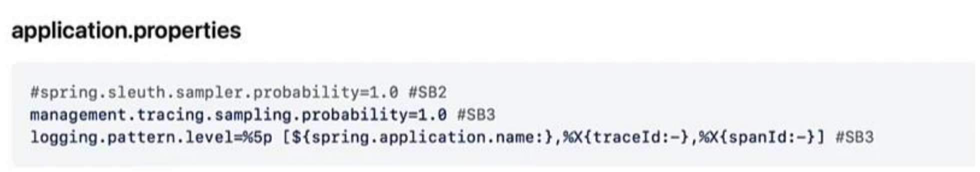


```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

- 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.

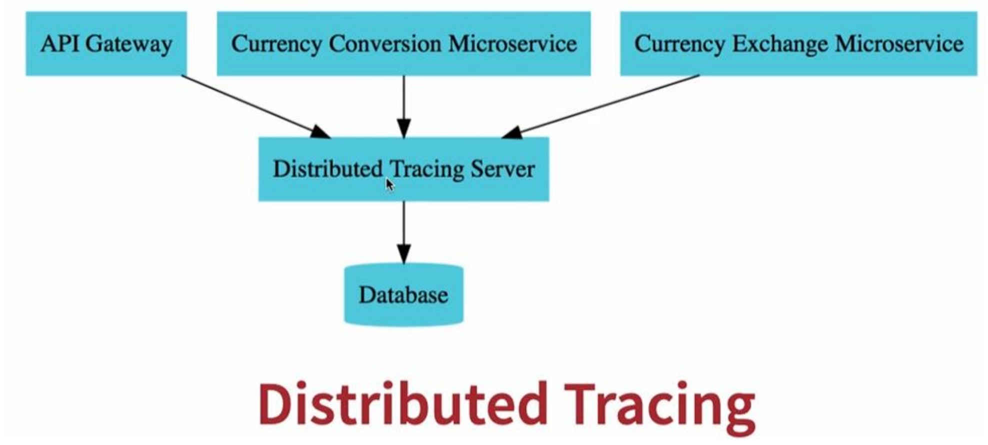


application.properties

```
#spring.sleuth.sampler.probability=1.0 #SB2
management.tracing.sampling.probability=1.0 #SB3
logging.pattern.level=%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}] #SB3
```

- 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.

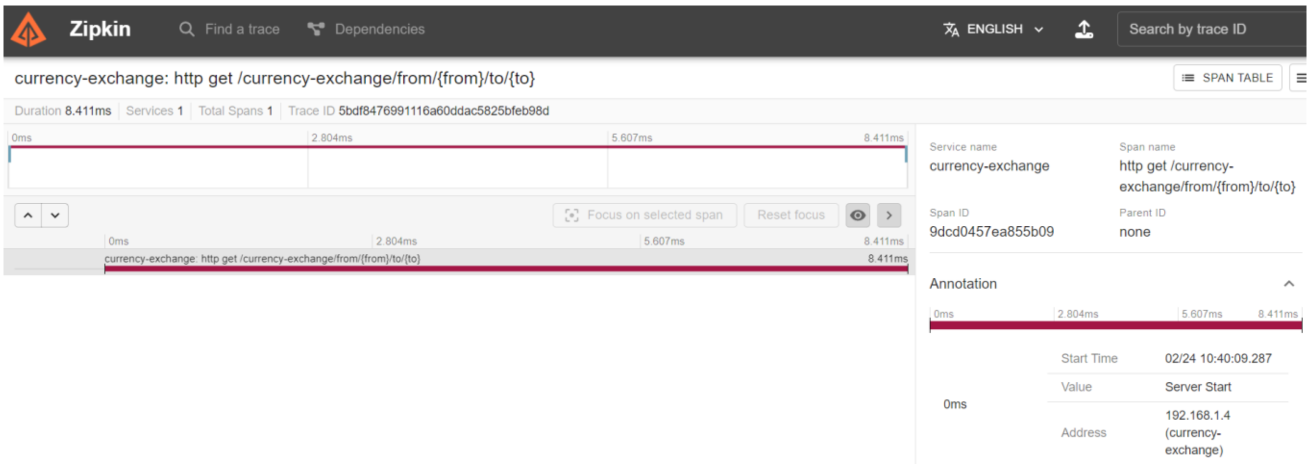
Adding Tracing configuration to Microservices


API Gateway
Currency Conversion Microservice
Currency Exchange Microservice
Distributed Tracing Server
Database
Distributed Tracing

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..)

```
2024-02-24T10:40:09.289+05:30 INFO [currency-exchange,5bdf8476991116a60ddac5825bfeb98d,9dcd0457ea855b09]
c.s.m.CurrencyExchangeController : retrieveExchangeValue called with USD to INR
```



currency-exchange: http get /currency-exchange/from/{from}/to/{to}

Duration 8.411ms | Services 1 | Total Spans 1 | Trace ID 5bdf8476991116a60ddac5825bfeb98d

0ms 2.804ms 5.607ms 8.411ms

currency-exchange: http get /currency-exchange/from/{from}/to/{to} 8.411ms

Service name: currency-exchange
Span name: http get /currency-exchange/from/{from}/to/{to}
Span ID: 9dcd0457ea855b09
Parent ID: none

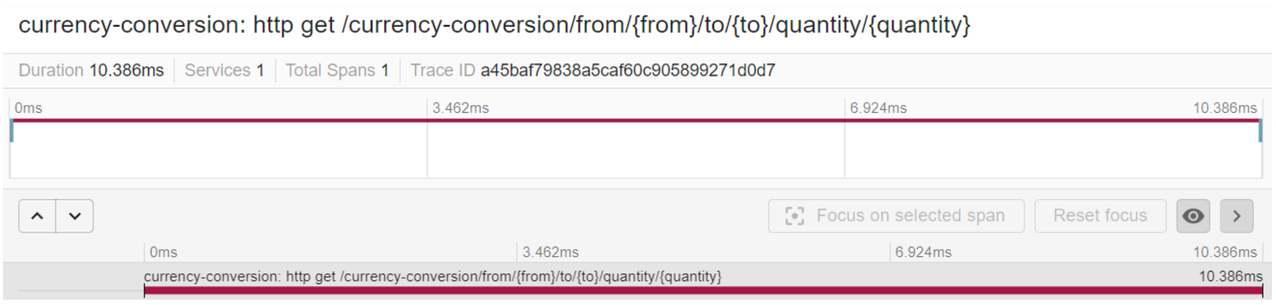
Annotation

0ms	Start Time	02/24 10:40:09.287
	Value	Server Start
	Address	192.168.1.4 (currency-exchange)

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



currency-conversion: http get /currency-conversion/from/{from}/to/{to}/quantity/{quantity}

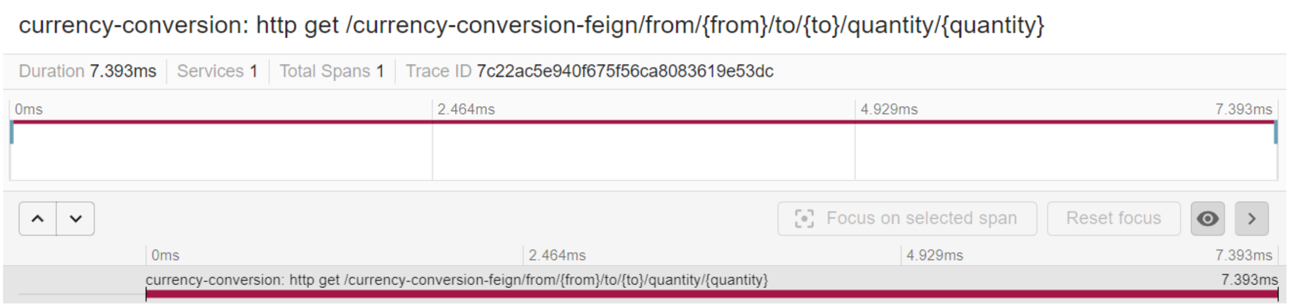
Duration 10.386ms | Services 1 | Total Spans 1 | Trace ID a45baf79838a5caf60c905899271d0d7

0ms 3.462ms 6.924ms 10.386ms

0ms 3.462ms 6.924ms 10.386ms

currency-conversion: http get /currency-conversion/from/{from}/to/{to}/quantity/{quantity} 10.386ms

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



currency-conversion: http get /currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}

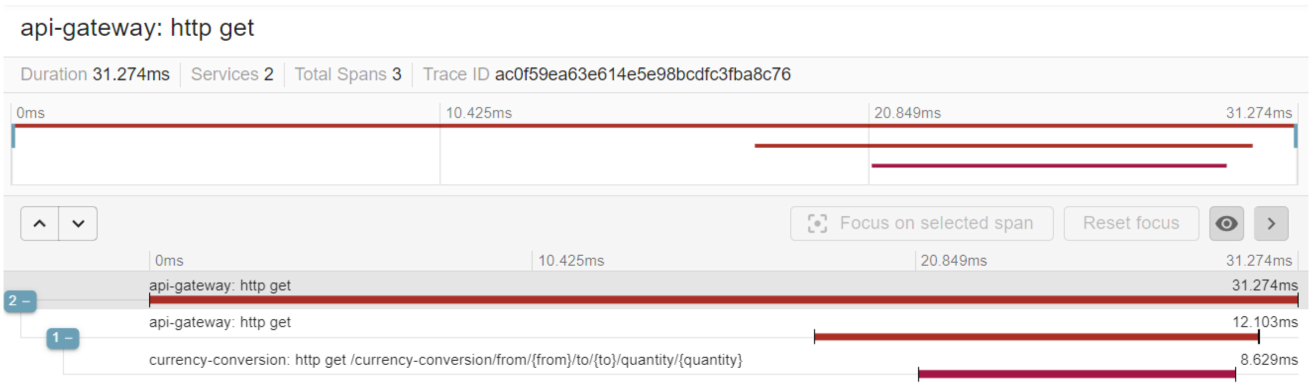
Duration 7.393ms | Services 1 | Total Spans 1 | Trace ID 7c22ac5e940f675f56ca8083619e53dc

0ms 2.464ms 4.929ms 7.393ms

0ms 2.464ms 4.929ms 7.393ms

currency-conversion: http get /currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity} 7.393ms

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



api-gateway: http get

Duration 31.274ms | Services 2 | Total Spans 3 | Trace ID ac0f59ea63e614e5e98bcdfc3fba8c76

0ms 10.425ms 20.849ms 31.274ms

0ms 10.425ms 20.849ms 31.274ms

2 – api-gateway: http get 31.274ms

1 – api-gateway: http get 12.103ms

currency-conversion: http get /currency-conversion/from/{from}/to/{to}/quantity/{quantity} 8.629ms

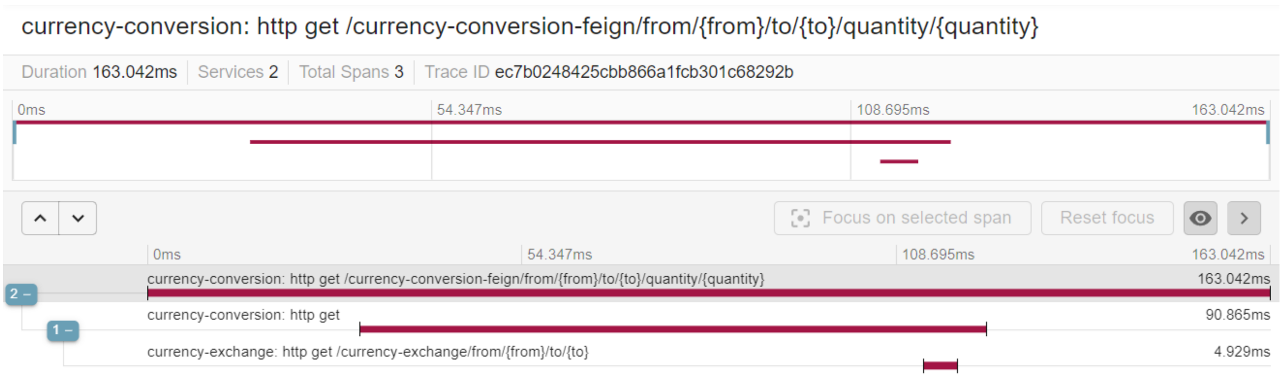
Problem: Here we can currency-conversion-ms is calling currency-exchange-ms using RestTemplate & Feign Client & we can see traces from API gateway to currency-conversion-ms, but the call to currency-exchange-ms, we're unable to trace.

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

```xml
<!-- COMMON CHANGES + -->
<!-- Enables tracing of REST API calls made using Feign - V3 ONLY-->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-micrometer</artifactId>
</dependency>
```

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

- We can see currency-exchange-ms trace now

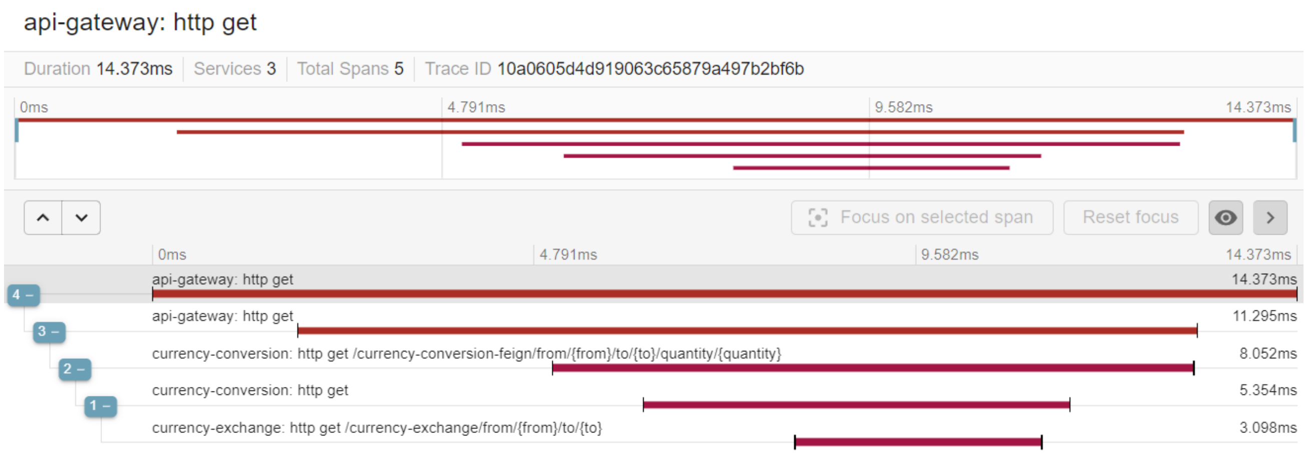


currency-conversion: http get /currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}

Duration 163.042ms | Services 2 | Total Spans 3 | Trace ID ec7b0248425cbb866a1fcb301c68292b

```
2 – currency-conversion: http get /currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}   163.042ms
  1 – currency-conversion: http get                                                                      90.865ms
        currency-exchange: http get /currency-exchange/from/{from}/to/{to}                                4.929ms
```

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



api-gateway: http get

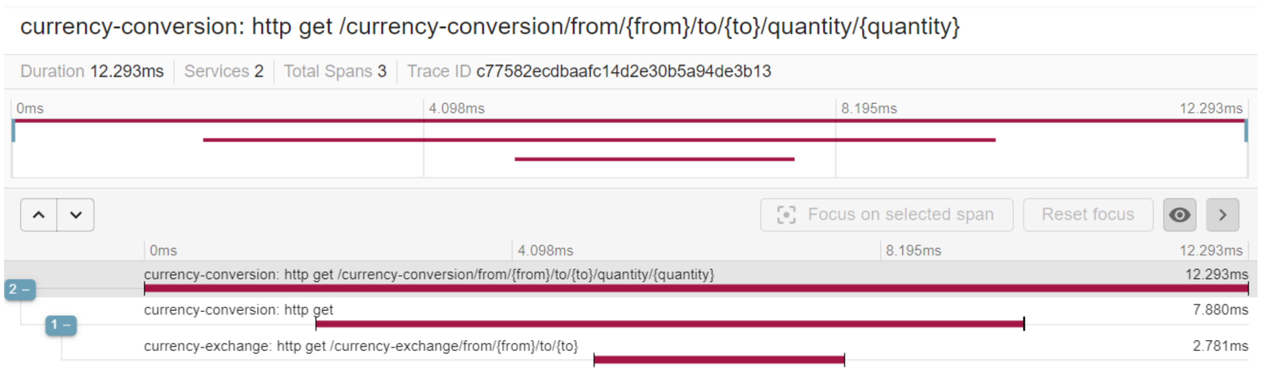
Duration 14.373ms | Services 3 | Total Spans 5 | Trace ID 10a0605d4d919063c65879a497b2bf6b

```
4 – api-gateway: http get                                                                                 14.373ms
  3 – api-gateway: http get                                                                               11.295ms
    2 – currency-conversion: http get /currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}   8.052ms
      1 – currency-conversion: http get                                                                    5.354ms
            currency-exchange: http get /currency-exchange/from/{from}/to/{to}                              3.098ms
```

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

```java
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
class RestTemplateConfiguration {

    @Bean
    RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
}

@RestController
public class CurrencyConversionController {

        @Autowired
        private CurrencyExchangeProxy proxy;

    @Autowired
    private RestTemplate restTemplate;

        @GetMapping("/currency-conversion/from/{from}/to/{to}/quantity/{quantity}")
        public CurrencyConversion calculateCurrencyConversion(
                        @PathVariable String from,
                uriVariables.put("from",from);
                uriVariables.put("to",to);

                //ResponseEntity<CurrencyConversion> responseEntity = new RestTemplate().getForEntity
                ResponseEntity<CurrencyConversion> responseEntity = restTemplate.getForEntity
                ("http://localhost:8000/currency-exchange/from/{from}/to/{to}",
                                CurrencyConversion.class, uriVariables);
```

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

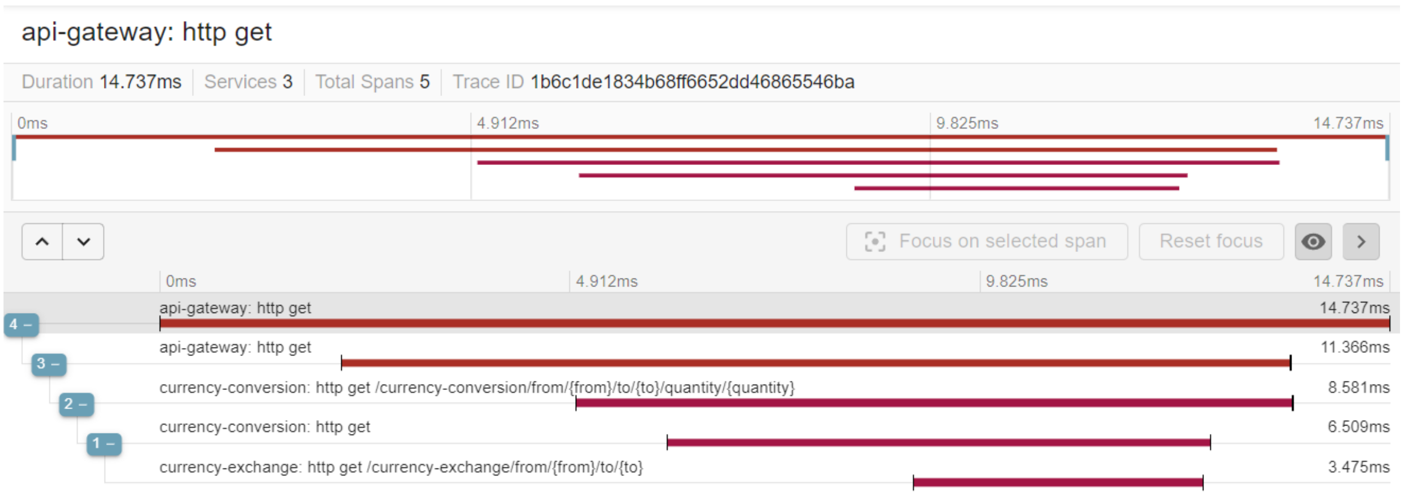


currency-conversion: http get /currency-conversion/from/{from}/to/{to}/quantity/{quantity}

Duration 12.293ms | Services 2 | Total Spans 3 | Trace ID c77582ecdbaafc14d2e30b5a94de3b13

```
2 – currency-conversion: http get /currency-conversion/from/{from}/to/{to}/quantity/{quantity}   12.293ms
1 –   currency-conversion: http get                                                              7.880ms
        currency-exchange: http get /currency-exchange/from/{from}/to/{to}                      2.781ms
```

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



api-gateway: http get

Duration 14.737ms | Services 3 | Total Spans 5 | Trace ID 1b6c1de1834b68ff6652dd46865546ba

```
4 – api-gateway: http get                                                                          14.737ms
3 –   api-gateway: http get                                                                        11.366ms
2 –     currency-conversion: http get /currency-conversion/from/{from}/to/{to}/quantity/{quantity}  8.581ms
1 –       currency-conversion: http get                                                             6.509ms
            currency-exchange: http get /currency-exchange/from/{from}/to/{to}                     3.475ms
```

Docker Deployment

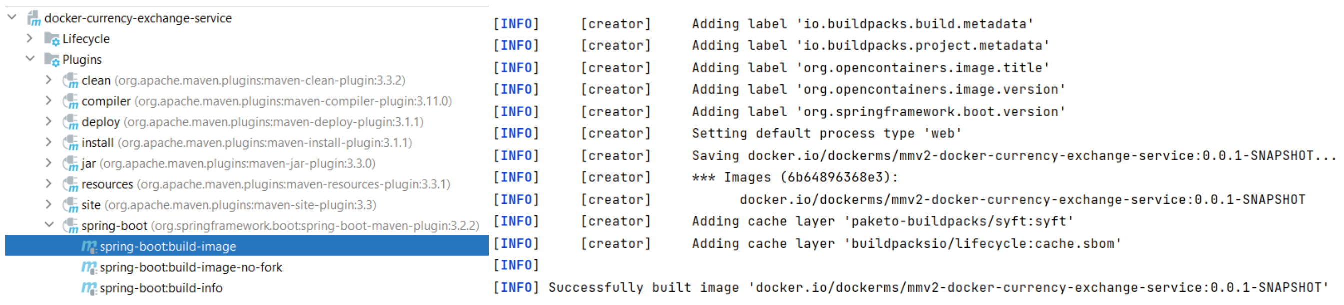
- Before Spring Boot version 2.3, there was a lot of process involved in creating a Docker container for Spring Boot application or Java applications.
- But from Spring Boot version 2.3, it’s very easy to create Docker containers.

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

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <image>
                    <name>dockerms/mmv2-${project.artifactId}:${project.version}</name>
                </image>
                <pullPolicy>IF_NOT_PRESENT</pullPolicy>
            </configuration>
        </plugin>
    </plugins>
</build>
```

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**


docker-currency-exchange-service
Lifecycle
Plugins
clean (org.apache.maven.plugins:maven-clean-plugin:3.3.2)
compiler (org.apache.maven.plugins:maven-compiler-plugin:3.11.0)
deploy (org.apache.maven.plugins:maven-deploy-plugin:3.1.1)
install (org.apache.maven.plugins:maven-install-plugin:3.1.1)
jar (org.apache.maven.plugins:maven-jar-plugin:3.3.0)
resources (org.apache.maven.plugins:maven-resources-plugin:3.3.1)
site (org.apache.maven.plugins:maven-site-plugin:3.3)
spring-boot (org.springframework.boot:spring-boot-maven-plugin:3.2.2)
spring-boot:build-image
spring-boot:build-image-no-fork
spring-boot:build-info

```
[INFO]     [creator]     Adding label 'io.buildpacks.build.metadata'
[INFO]     [creator]     Adding label 'io.buildpacks.project.metadata'
[INFO]     [creator]     Adding label 'org.opencontainers.image.title'
[INFO]     [creator]     Adding label 'org.opencontainers.image.version'
[INFO]     [creator]     Adding label 'org.springframework.boot.version'
[INFO]     [creator]     Setting default process type 'web'
[INFO]     [creator]     Saving docker.io/dockerms/mmv2-docker-currency-exchange-service:0.0.1-SNAPSHOT...
[INFO]     [creator]     *** Images (6b64896368e3):
[INFO]     [creator]           docker.io/dockerms/mmv2-docker-currency-exchange-service:0.0.1-SNAPSHOT
[INFO]     [creator]     Adding cache layer 'paketo-buildpacks/syft:syft'
[INFO]     [creator]     Adding cache layer 'buildpacksio/lifecycle:cache.sbom'
[INFO]
[INFO] Successfully built image 'docker.io/dockerms/mmv2-docker-currency-exchange-service:0.0.1-SNAPSHOT'
```

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:
```