Microservices with SpringBoot, REST APIs, Docker, and Kubernetes:

Webservice:

1. Software system designed to support interoperable(Not platform dependent) machine-to-machine interaction over a network.
2. Data exchange takes place as request and response (request and response is also platform independent).
3. We can use the webservice platform independent with the help of XML and JSON object.

Service Definition –

Every webservice offers a service definition which contains structure of the webservices as below:

1. Request/ Response Format
2. Request Structure
3. Response Structure
4. Endpoint

Request - Input to webservice.
Response - output from the webservice
Message Exchange Format - XML or JSON

Service Provider - Is the one which provides the webservice.
Service Consumer - Consumer is the one which consumes the webservice.
Service Definition - Is the contract between Service provider and the service consumer.

Transport - Defines how a service is called. Transport indicates weather if the webservice is exposed in the web or queue. Such as HTTP and Messaging Queue(MQ).

Web Service Group

1. SOAP BASED
2. REST

SOAP and REST are not comparable.

SOAP - Simple Object access protocol

Soap defines a specific way of building web services.

1. It uses XML.
2. Soap defines a specific request and response structure SOAP- ENV: EnvelopeSOAP -ENV: Header SOAP -ENV: Body
3. Soap transport support both HTTP or MQ transport.
4. Soap service definition is WSDL : Web service definition Language.

WSDL - Endpoint - All Operations - Request Structure - Response Structure

REST - Representational State Transfer

1. Rest uses HTTP(Hypertext transfer protocol)
2. HTTP Request Methods are - GET, POST, PUT, PATCH, DELETE
3. HTTP Response Status code - 200, 404, 201

REST KEY ABSTRACTION - Resource

1. A resource has an URI (Uniform Resource Identifier) - /trracs/employee/1 , /trracs/employee, /trracs/employee/employeee
2. Resource can be XML, JSON, XML.
3. No restriction on Data Exchange format but JSON is popular
4. Transport supports only HTTP
5. Service Definition is not standard. WADL, swagger are kind of popular sources.

Why SpringBoot?

@SpringBootApplication
@RestController
@Controller
@GetRequestMapping
@RequestMapping
@ResponseBody
@GetMapping
@PostMapping
@PutMapping
@DeleteMapping
@PathVariable
@RequestBody
@ControllerAdvice
@ExceptionHandler
@Valid

RESPONSE:

404 - Resource not found
400 - validation error(Bad request)
500 - Server Error or Exception
200 - Success.
201 - Created
204 - No content
401 - Unauthorized

@ResponseStatus(code = HttpStatus.*NOT_FOUND*)

Background:

Dispatcher Servlet is mapping to URL root.
Our Requests are getting handled by dispatcher servlet which is called front controller pattern.

1. **Mappings Servlets:** Once request hits the dispatcher servlet, it tries to map the request to the respective controller.

2. **Auto configuration:** DispatcherServletAutoConfiguration (spring automatically configures this)

3. **How does Objects converted to JSON? :**

 i. **@ResponseBody + JacksonHTTPMessageConvertersConfiguration**
    Which concludes that The Java Beans I.e. objects are converted into Messages with the help of @RequestBody annotation and JacksonHTTPMessageConvertersConfigurations automatically.

4. **Error Mapping:** Errors are also handled explicitly by Auto configuration which is ErrorMvcAutoConfiguration.

5. **How are all Jars available?(Spring, Spring MVC, Jackson, Tomcat)**
StarterProjects - SpringBootStarterWeb contains all the dependancies.Spring Boot takes care of all the above. And the two main features are start projects and auto configuration.

@PathVariable - used for passing parameters

GET - Retrieve details of a resource.
POST - Create a new resource.
PUT - Update an existing resource.
PATCH - Update part of a resource.
DELETE - Delete a resource.

BEST PRACTICE IS TO ALWAYS USE PRURAL'S in URI

/employees
/employees/1
/employees/1/posts

ResponseEntityExceptionHandler handles all the exceptions by default.

1. Get Mapping Example:

```
// getting a specific user with id.
@GetMapping("/users/{id}")
public User getUser(@PathVariable int id){
   User user = userService.getUserId(id);
if(user == null){
    System.out.println(id);
    throw new UserNotFoundException("ID:"+id);
}
return user;

}

//service
@Override
public User getUserId(int id) {
    Predicate<? super User> predicate = user -> user.getId() == id;
//handling null with Else null
    return userList.stream().filter(predicate).findFirst().orElse(null);
}
```

2. Post Mapping with Response Entity and returning the URI location within the Request:

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody User user){

   User savedUser =  userService.createUser(user);
    URI location = ServletUriComponentsBuilder.
            fromCurrentRequest().
            path("/{id}").
            buildAndExpand(savedUser.getId()).
            toUri();
     return ResponseEntity.created(location).build();
}
```

3. Delete Mapping with id

```
@DeleteMapping(path = "/users/{id}")
public void deleteUserById(@PathVariable int id){
        userService.deleteUserById(id);
}
```

Documentation

1. **Swagger** - Swagger Specification and Swagger Tools.
2. **OpenAPI** - OPENAPI specification was created based on Swagger Specification.

Swagger Tools - Swagger UI (Visualize and interact with REST API)
OpenAI Specification - Standard, language-agnostic interface.

Content Negotiation

Same Resource - Same URI
But different representations are possible such as

1. Different Content type - XML or JSON
2. Different Language - English or Dutch

In content negotiation, service provider and consumer negotiate with each other saying what content type and language is needed.

com.fasterxml.jackson.dataformat. — pom helps us to change the data format in xml or json.

Internationalization - i18n

As part of i18n HTTP Request header is used with Accept-Language tag.

En- English
Nl -Dutch

We make use of **message.properties** file for the property with respect to languages and then Locale class.

```
Locale locale = LocaleContextHolder.getLocale();
return messageSource.getMessage("good.morning.message",null,"defaultmessage",locale);
```

Versioning

Whenever we want to enhance our application it's better to maintain different versions. Which will give the flexibility to consumer to use different versions.

Versioning can be done via:

1. **URI - changing the URI : (Twitter)**
http://localhost:8080/persons/v1/persons
http://localhost:8080/persons/v2/persons

2. Request Parameters (Amazon)
http://localhost:8080/persons/person?version=1
http://localhost:8080/persons/person?version=2
@GetMapping(path = "/person", params ="version=1")
@GetMapping(path = "/person", params ="version=2")

3. Request Header (Microsoft)
http://localhost:8080/persons/person/header
@GetMapping(path = "/person/header", headers ="X-API-VERSION=1")
@GetMapping(path = "/person/header", headers ="X-API-VERSION=2")

4. Media type (Accept Header) (Github)
http://localhost:8080/persons/person/accept
@GetMapping(path = "/person/accept", produces ="application/vnd.company.app-v1+json")

Recommended Approach for Versioning:

1. **URI Pollution** - URI versioning and Request Parameter Versioning.
2. **Misuse of HTTP Headers** - Headers versioning and Media Type versioning.
3. **Caching** - Headers versioning and Media type versioning cannot be cached with help of URL. We also have to look at headers while caching.
4. **Can we execute the request on the browser?** URI and Request parameters can be executed but not others.
5. **API Documentation**

HATEOAS

HATEOAS - Hypermedia as the Engine of Application State

Website : Data can be viewed, and actions can be performed in any websites.

Enhancing REST API to give data and perform actions on the resources we can use HATEOAS.

Implementation of HATEOS:

1. Customer format and Implementation:
2. Use Standard Implementation : **HAL** (JSON Hypertext Application Language): Simple format that gives a consistent and easy way to hyperlink between resources in API.

Spring HATEOAS: Generates HAL responses with hyperlinks to resources.

```
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;

@GetMapping("/usersmodel/{id}")
public EntityModel<User> getUsers(@PathVariable int id) throws UserNotFoundException{

    User user = userService.getUserId(id);
    if(user == null){
        System.out.println(id);
        throw new UserNotFoundException("ID:"+id);
    }
    EntityModel<User> model = EntityModel.of(user);
    WebMvcLinkBuilder builder = linkTo(methodOn(this.getClass()).getUserList());
    model.add(builder.withRel("all-users"));
    return model;
}
```
Output:
```
{
    "id": 1,
    "userName": "Vinoth",
    "birthDate": "1994-03-26",
    "_links": {
        "all-users": {
            "href": "http://localhost:8080/usersapi/users"
        }
    }
}
```

Implementing Static Filtering for REST API:.

Serialization: Convert object to stream (Ex: JSON) most popular JSON in java is Jackson.

Customizing REST API response returned by Jackson framework:

1. Customizing field name in response:

@JSONProperty (Used to customize attribute and element name in response)

```
@JsonProperty("user_name")
     private String userName;
      Response:{
       "id": 1,
       "birthDate": "1994-03-26",
       "user_name": "Vinoth"
   }
```

2. Return only selected fields:

Filtering:

Filter out password

Two types of Filtering:

1. Static Filtering on Bean

Same filtering for a bean across different REST API using **@JsonIgnoreProperties, @JsonIgnore**

Class Level: **@JsonIgnoreProperties("field1")**
public class SomeBean

Attribute Level: **@JsonIgnore**
private String field1;

2. Dynamic Filtering

Customize filtering for a bean for specific REST API
- **@JsonFilter** with FilterProvider

MappingJacksonValue: Allows us to add serialization logic along with data.

```
    @GetMapping("/filtering")
  public MappingJacksonValue filtering(){
    SomeBean someBean = new SomeBean("Vinoth","Kumar","Jayandiran");
    MappingJacksonValue mappingJacksonValue = new MappingJacksonValue(someBean);
    SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.filterOutAllExcept("field1");
    FilterProvider filters = new SimpleFilterProvider().addFilter("SomeBeanFilter", filter);
    mappingJacksonValue.setFilters(filters);
    return mappingJacksonValue;
}

@JsonFilter("SomeBeanFilter")
public class SomeBean
```

Spring Actuator: localhost:8080/actuator

Actuator is a starter dependency. Which helps to manage and monitor application in Production.

to expose all the api’s property to be configured in application.properties
management.endpoints.web.exposure.include=*

actuator/beans
actuator/metrics
actuator/mappings
actuator/env
actuator/health
actuator/heapdump
actuator/threaddump

// REST API using HAL Explorer

1. HAL - (JSON Hypertext Application Language)Simple format that gives a consistent and easy way to hyperlink between resources in API.
2. HAL ExplorerAn API explorer for RESTful Hypermedia APIs using HALHAL Explorer enables HAL explorer to use APIs with ease
3. Spring Boot HAL explorerAuto-configures HAL Explorer for Spring Boot Projects.spring-data-rest-hal-explorer

Microservices With Spring Cloud:

Microservices :

It’s a style of developing a single application as a suite of small services each running its own process communicating with very lightweight mechanisms often an HTTP resource API.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

Importantance for Microservices:

1. **REST**
2. **Small Well Chosen Deployable Units**
3. **Cloud Enabled**

CHALLENGES:

1. Bounded Context - What should to do, and what not to do (This is Evolutionary Decision)
2. Configuration Management - 100’s of services, lot of management.
3. Dynamic scale up and down - Load distribution, scaling up and down.
4. Visibility - Which services are in defect.
5. Pack Of Cards - If not well designed then it’s a mess. If one service fails it’s a waste.

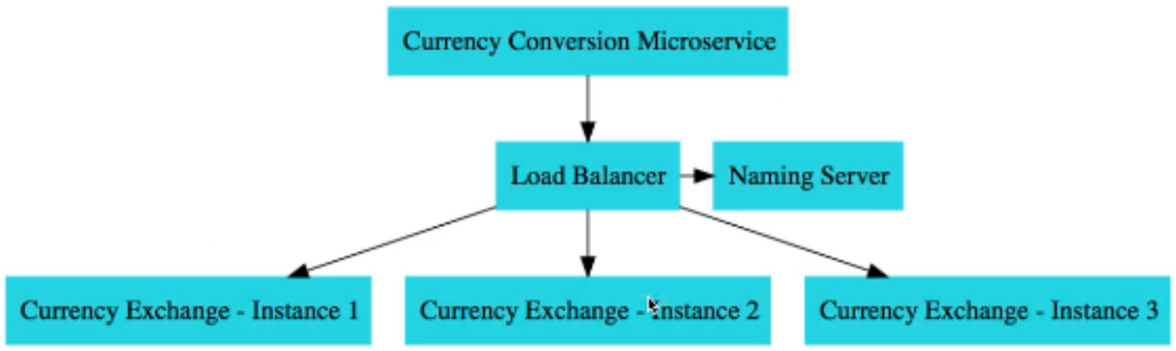
// Spring Cloud:

Multiple Microservices with multiple environments which means more configurations.

Configurations of different microservices can be stored in SpringCloudConfigServer.

SpringCloudConfigServer provides an approach where we can store all configurations in a GIT repository.

SpringCloudConfigServer can be used to expose the configuration to all the microservices.


Currency Conversion Microservice
Load Balancer
Naming Server
Currency Exchange - Instance 1
Currency Exchange - Instance 2
Currency Exchange - Instance 3

Suppose there is a service A which uses multiple instances of a service B. It’s possible that one or more service of service B can be removed or added.

ServiceA should be able to distribute the load between all the instances of the service .

The Solution will be using the naming Server I.e **Eureka**

DYNAMIC SCALE UP AND DOWN

1. **Naming Server (Eureka)**
    1. **Service Registration -** All the instances of all microservices will register with naming server.
    2. **Service Discovery -** Service A will enquire the Eureka server to give the active instances of Service B, Eureka helps to establish dynamic relation between Service A and B.

2. **Ribbon (Client Side Load Balancing) -** Service A will host Ribbon. Which will help to distribute the load between existing instances of Service B. Latest is **Spring Cloud Load Balancer**

3. **Feign (Easier REST clients) -** It will be used by Service A to write simple Restful webservice.

VISIBILITY AND MONITORING:

1. **Zipkin Distributed Tracing**
2. **Netflix Zull API Gateway**. Latest is **Spring cloud Gateway.**

FAULT TOLERANCE:

Hystrix. Latest is **Resilience4j.**

Advantages Of Microservices:

1. Enables to adapt **new technologies and processes easily**.
2. Each Microservices can be with different technologies since it communicates with each other with message.
3. Microservices are **dynamic scaling.**
4. **Faster release cycles**

Microservices components:

1. **Ports**
2. **URLS**

Config Server:

@EnableConfigServer - at **ApplicationServer** class after @SpringBootApplication

Configuring git repo in applicationconfig.properties

```
server.spring.cloud.config.server.git.uri=file:///Users/vinoth/git-localconfig-repo
```

Connecting Microservice with config server.

1. **Configuration in pom.xml** - Config Client
2. **URL** -

- # **Spring cloud** starter config, we must configure how SpringCloud starter config connects to spring cloud.
- **spring.config.import=optional:configserver:http://localhost:8888**

Centralized Configuration

If there are multiple environments for a service, to separate the configuration and have it centralized helps.

```
spring.profiles.active=dev
spring.profiles.active=qa
spring.profiles.active=prod
```

// connecting application to a in-memory database:
Pom.xml:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
</dependency>
```

application.properties:

```
#to show the queries.
spring.jpa.show-sql=true
#url to connect to in memory h2 database
spring.datasource.url=jdbc:h2:mem:testdb
#to enable the h2 console
spring.h2.console.enabled=true
```

#By default the data stored in data.sql file under java/resources is loaded first before creating tables. To load the data after the tables are created we use below property

```
spring.jpa.defer-datasource-initialization=true
```

Calling a rest service within a rest service

METHOD 1:

```
@GetMapping("/currency-conversion/from/{from}/to/{to}/quantity/{quantity}")
public CurrencyConversion calculateCurrencyConversion(@PathVariable String from,
                                                      @PathVariable String to,
                                                      @PathVariable BigDecimal quantity){

    HashMap<String, String> uriVariables = new HashMap<>();
    uriVariables.put("from",from);
    uriVariables.put("to",to);

    ResponseEntity<CurrencyConversion> responseEntity = new
RestTemplate().getForEntity("http://localhost:8000/currency-exchange-
service/from/{from}/to/{to}"
            ,CurrencyConversion.class, uriVariables);
    CurrencyConversion currencyConversion = responseEntity.getBody();
    return new CurrencyConversion(currencyConversion.getId(),from,to,
            quantity, currencyConversion.getConversionMutiple(),
            quantity.multiply(currencyConversion.getConversionMutiple()), "0006");
}
```

METHOD 2:

Instead of writing the complex code each and every time, Spring cloud provides a service called as **FEIGN.**

// In currencyConversionService

Pom dependency:

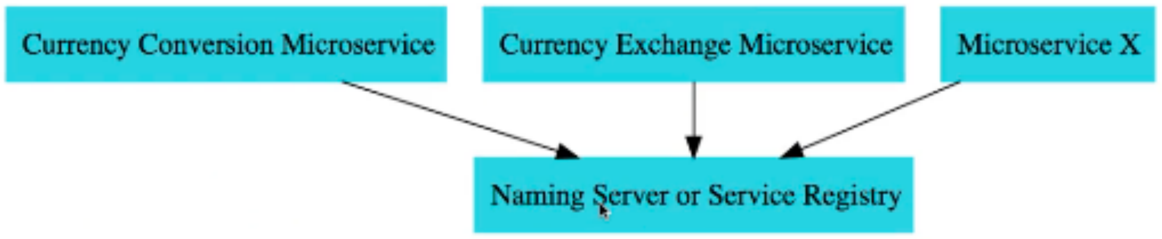
```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableFeignClients
public class CurrencyConversionServiceApplication {

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

}
```

```
@FeignClient(name="currency-exchange-service", url="localhost:8000")
public interface CurrencyExchangeProxy {

   @GetMapping("/currency-exchange-service/from/{from}/to/{to}")
   public CurrencyConversion retrieveExchangeValue(@PathVariable String from,
                                                   @PathVariable String to);
}
```

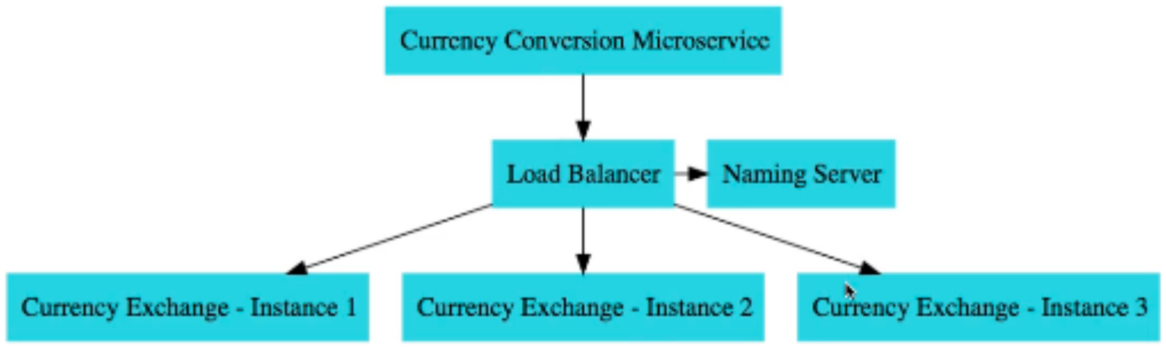
```
@GetMapping("/currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}")
public CurrencyConversion calculateCurrencyConversionFeign(@PathVariable String from,
                                                           @PathVariable String to,
                                                           @PathVariable BigDecimal quantity){


   CurrencyConversion currencyConversion =
currencyExchangeProxy.retrieveExchangeValue(from, to);
   return new CurrencyConversion(currencyConversion.getId(),from,to,
         quantity, currencyConversion.getConversionMutiple(),
         quantity.multiply(currencyConversion.getConversionMutiple()), "0006");
}
```

@FeignClient(name="currency-exchange-service", url="localhost:8000")
Instead of hardcoding the URL from multiple services without knowing their current state (If it went down or up), to overcome this we go for **Service Registry or Naming Server.**


Currency Conversion Microservice
Currency Exchange Microservice
Microservice X
Naming Server or Service Registry

Naming Server

In a Microservices Architecture all the microservices would register with the **Service Registry.**


Currency Conversion Microservice
Load Balancer
Naming Server
Currency Exchange - Instance 1
Currency Exchange - Instance 2
Currency Exchange - Instance 3

If Service A wants to talk to Service B then it will ask the Service Registry the address of Service B.

Service Registry or Naming Server. (Eureka)

Dependencies:

DevTools, Actuator, Eureka Server

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

```
@EnableEurekaServer
@SpringBootApplication
public class NamingServerApplication {

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

}
```

//application.properties

```
server.port=8761

#to avoid naming server self-register itself.
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

To register the microservices with Naming Server:

1. Edit pom.xml in Service A and B

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

Then after restarting the Microservices, it will be automatically added in the Eureka Naming Server.

Just to be safe we can configure the naming service URL in **application.properties.**

```
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
```

Load Balancing with Eureka, Feign, & Spring Cloud LoadBalancer:

Total number of changes 2.

1. **ApplicationClass**
2. **ProxyClass**

1. ApplicationClass

```
@SpringBootApplication
@EnableFeignClients
public class CurrencyConversionServiceApplication {

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

}
```

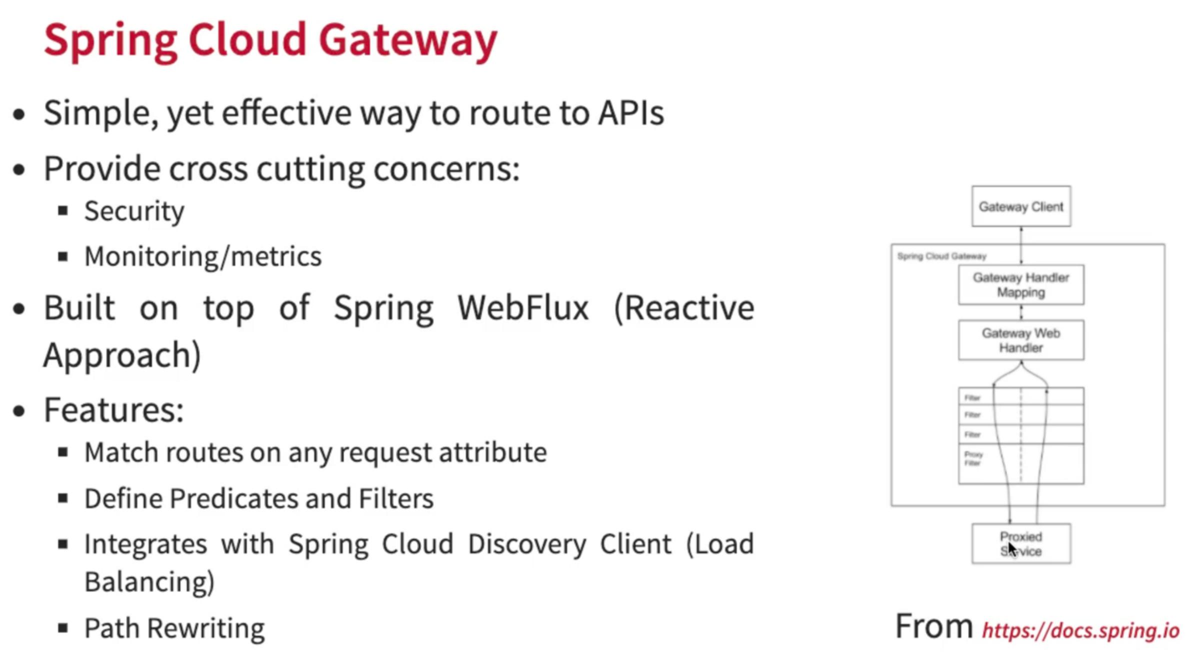
2. Proxy class

```
@FeignClient(name="currency-exchange-service")
public interface CurrencyExchangeProxy {

@GetMapping("/currency-exchange-service/from/{from}/to/{to}")
   public CurrencyConversion retrieveExchangeValue(@PathVariable String from,
                                    @PathVariable String to);
}
```

That’s it.

Now Load Balancing is happen with active instances.

Spring Cloud API Gateway


Gateway Client
Spring Cloud Gateway
Gateway Handler Mapping
Gateway Web Handler
Filter
Filter
Filter
Proxy Filter
Proxied Service

Spring Cloud Gateway

- Simple, yet effective way to route to APIs
- Provide cross cutting concerns:
  - Security
  - Monitoring/metrics
- Built on top of Spring WebFlux (Reactive Approach)
- Features:
  - Match routes on any request attribute
  - Define Predicates and Filters
  - Integrates with Spring Cloud Discovery Client (Load Balancing)
  - Path Rewriting

From *https://docs.spring.io*

Dependencies:

DevTools, Actuator, Eureka Discovery Client, Gateway

Pom.xml:

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

```
//application.properties
#registering API gateway with eureka naming server (Not Manditory)

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

#enabling service discovery using client. It helps API Gateway to talk to different services
registered with Eureka server.
spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

There will be 100’s of Microservices which will have common features like Authentication, Authorization, Logging, Rate Limiting, Security, Monitoring/ metrics, and resiliency. The solution to go is **API Gateway**. The old one is **Zool**.

Spring Cloud API Gateway.

Now with the help of API Gateway port, we can use navigate to any service which is registered in Eureka naming server(Service registry).

Currency Exchange Service

http://localhost:8000/currency-exchange-service/from/USD/to/INR
http://localhost:8001/currency-exchange-service/from/USD/to/INR

Currency Conversion Service

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

#feign

http://localhost:8100/currency-conversion-feign/from/KWR/to/INR/quantity/13

Eureka Naming Server(Service Registry)

http://localhost:8761/eureka

Api Gateway URL:

http://localhost:8765/currency-exchange-service/currency-exchange-service/from/USD/to/INR

http://localhost:8765/currency-conversion-service/currency-conversion-feign/from/KWR/to/INR/quantity/13

So, instead of typing the entire URL each and every time, we can route the URL in API gateway using **CustomFilters** as below.

```
@Configuration
public class APIGatewayConfiguration {

   @Bean
   public RouteLocator gatewayLocator(RouteLocatorBuilder routeLocatorBuilder){
      return routeLocatorBuilder.routes().route(p ->
            p.path("/get").filters(f->f.addRequestHeader("MyHeader","MyURI"))
            .uri("http://httpbin.org:80"))
            .route(p -> p.path("/currency-exchange-service/**").uri("lb://currency-exchange-
service"))
            .route(p -> p.path("/currency-conversion/**").uri("lb://currency-conversion-service"))
            .route(p -> p.path("/currency-conversion-feign/**").uri("lb://currency-conversion-
service"))
            .route(p -> p.path("/currency-conversion-new/**")
            .filters(f->f
            .rewritePath("/currency-conversion-new/(?<segment>.*)",
             "/currency-conversion-feign/${segment}")).uri("lb://currency-conversion-service"))
.build();
   }
}
```

http://localhost:8765/currency-exchange-service/from/USD/to/INR
http://localhost:8765/currency-conversion-feign/from/KWR/to/INR/quantity/13
http://localhost:8765/currency-conversion/from/KWR/to/INR/quantity/13
http://localhost:8765/currency-conversion-new/from/KWR/to/INR/quantity/13

Global Filters

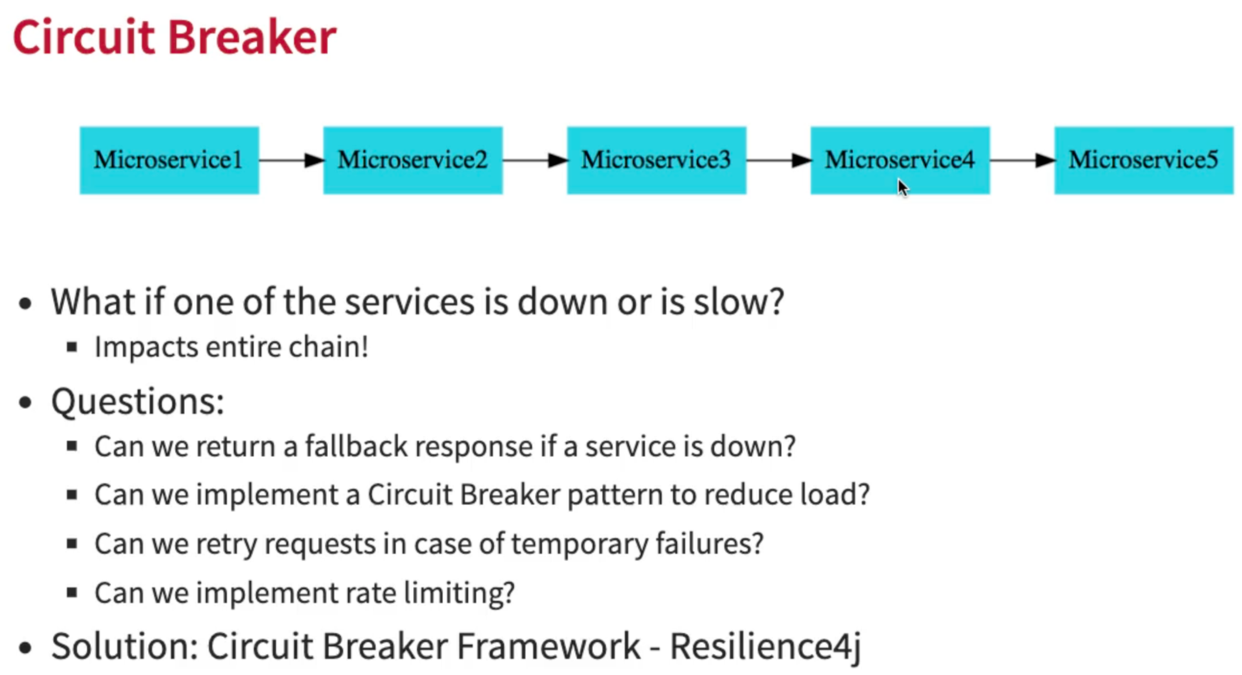
Let’s say we want to log all the request that goes through API Gateway.

```
@Component
public class LoggingFilter implements GlobalFilter {

  private final Logger logger = LoggerFactory.getLogger(LoggingFilter.class);
   @Override
   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
      logger.info("Path of the request received -> {}",exchange.getRequest().getPath());
      return chain.filter(exchange);
   }
```

Log:

```
[api-gateway] [ctor-http-nio-3] c.g.m.apigateway.filters.LoggingFilter   : Path of the request received -> /currency-conversion/from/KWR/to/INR/quantity/13
```

Circuit Breaker:



Circuit Breaker

- What if one of the services is down or is slow?
  - Impacts entire chain!
- Questions:
  - Can we return a fallback response if a service is down?
  - Can we implement a Circuit Breaker pattern to reduce load?
  - Can we retry requests in case of temporary failures?
  - Can we implement rate limiting?
- Solution: Circuit Breaker Framework - Resilience4j

resilience4j

#Retry attempts
resilience4j.retry.instances.sample-api.maxAttempts=5

#Retry wait time for each attempt
resilience4j.retry.instances.sample-api.waitDuration=1s

#response takes more time for sequent attempts. ex attempt 1 1sec, attempt 2 2sec, attempt 4 sec... exponentially increasing.
resilience4j.retry.instances.sample-api.enableExponentialBackoff=true

```
@GetMapping("/sample-api")
@Retry(name="sample-api", fallbackMethod = "hardCodedMethod")
public String sampleAPI(){
   logger.info("sample api call received");
   ResponseEntity<String> responseEntity= new
RestTemplate().getForEntity("http://localhost8080/helloApi",String.class);
   return responseEntity.getBody();
}
public String hardCodedMethod(Exception ex){
   return "fallback-response";
}
```

The code will try to reach the sample-api for 5 attempts as the retry is given as 5. Then it will return the fallbackMethod.

//Circuit Breaker

```
@GetMapping("/sample-api")
@CircuitBreaker(name="sample-api", fallbackMethod = "hardCodedMethod")
public String sampleAPI(){
   logger.info("sample api call received");
   ResponseEntity<String> responseEntity= new
RestTemplate().getForEntity("http://localhost8080/helloApi",String.class);
   return responseEntity.getBody();
}
public String hardCodedMethod(Exception ex){
   return "fallback-response";
}
```

application.properties.
#Only if 90% of requests fail then I want to switch to open state.
resilience4j.circuitbreaker.instances.default.failureRateThreshold =90

Circuit Breaker is a design pattern used to handle faults and failures in distributed systems.

It works by wrapping a protected function call (such as an API request to another microservices) in a circuit breaker object.

The circuit breaker pattern helps prevent cascading failures and improves system resilience by providing mechanisms such as

Failure detection, State Management, Fallback Mechanisms, Automatic Recovery

If a microservice is down, if all the call is failing so instead of calling the service and add load to the service the circuit breaker breaks the circuit and directly return a response.

How Will we know if the Microservices is up again?**States of Circuit Breaker**

1. Closed - When Circuit Breaker is calling a Microservices continuously. It will always be calling the Microservice.

2. Open - Circuit Breaker will not call the Microservices, it will directly return the fallback response.
3. Half_open - Circuit Breaker will be sending a percentage of requests to the microservice and for the rest it will return the fallback response.

When will Circuit Breaker Switch states?

1. When application is up initially Circuit Breaker will be in a closed state.
2. Then, when we are calling the dependent microservice and if 90% of requests are failing then Circuit Breaker switches to open state.
3. Once the Circuit Breaker is in open state it waits for a duration, which is configurable, after that the circuit breaker will switch to the half_open.
4. During the half_open state it tries and send the request it see if the Microservices is up. It sends a percentage of requests to the microservices and sees if it lets the response or not. Percentage is configurable. I Microservices is up and running it changes the state to closed and if not, it changes the state to open.

RateLimiting

application.properties

#2 request in every 10 seconds.
resilience4j.ratelimiter.instances.default.limitForPeriod=2

#duration in which number of requests should be processed.
resilience4j.ratelimiter.instances.default.limitRefreshPeriod=10s

```
@GetMapping("/sample-api")
@RateLimiter(name="sample-api")
public String sampleAPI(){
   logger.info("sample api call received");

   return "sample-api";
}
```

#10s => 10000 calls to the sample api.
Rate limiting is defining the duration between which number of requests would be processed.

BulkHead

Configuring how many concurrent calls are allowed for each api is called **bulkhead.**

application.properties:

```
#max of 10 concurrent calls.
resilience4j.bulkhead.instances.sample-api.maxConcurrentCalls=10
@GetMapping("/sample-api")
@Bulkhead(name="sample-api")
public String sampleAPI(){
   logger.info("sample api call received");
   return "sample-api";
}
```

Docker:

Commands:

#docker command we are making the internal port 5000 to host port 5000 and running a container
docker run -p 5000:5000 in28min/todo-rest-api-h2:1.0.0.RELEASE

#running image in detach mode.
docker run -p 5000:5000 -d in28min/todo-rest-api-h2:1.0.0.RELEASE

#Shows the logs.
docker logs XXXXXX(dockerId)

#Start tailing the logs to keep track.
docker logs -f XXXXXX(dockerId)

#ctrl c to stop tailing the logs.

#running the same image in another container with different port number in detached mode.
docker run -p 5001:5000 -d in28min/todo-rest-api-h2:1.0.0.RELEASE

#list the running containers
docker contaniner ls

#list all containers both running and stopped.
docker contaniner ls -a

#list docker images locally
docker images

#stop running container.
 docker container stop id(unique 5 digits not entire id)

#this will update the image and insert a new image with same image id but different tag
docker tag in28min/todo-rest-api-h2:1.0.0.RELEASE in28min/todo-rest-api-h2:latest

#To check all the steps involved in creating the image:docker image history imageId
docker image inspect imageId - Used to see entry point and in-depth details.

#To remove image from local:
 docker image remove imageId

#mysql is an official image. I.E
docker pull mysql

#Pause a container
docker container pause imageId

#unpause a container
docker container unpause imageId

#Inspect a container (Metadata, current status , platform, port binding, volume)
docker container inspect imageId

#stops a container (Does the graceful shutdown)
docker container stop imageId

```
#kills the container immediately
 docker container kill imageId

#prune will remove all the stoped containers.
 docker container prune

#starting container with restart mode
docker run -p 5000:5000 -d —restart=always in28min/todo-rest-api-
h2:1.0.0.RELEASE

#top process being done in a container
docker top imageId

#to show all the stats regarding the container it is running
docker stats imageId

#running container with specified memory and cpu quota
docker run -p 5001:5000 -m 512m —cpu-quota 5000 d —restart=always
in28min/todo-rest-api-h2:1.0.0.RELEASE

#df shows images, container, local volumes.
docker system df
```

Each microservices can be developed in different languages such as Java, python, go, JS etc..

To deploy multiple micro-services irrespective of the language or the framework is through containers.

1. **Create Docker images for each of the microservices.** **Docker** image contains everything a Microservices needs to run such as Application runtime, application code, and dependencies.

Docker containers can be run the same way in any infrastructure such as local machine, corporate data center, cloud infrastructure.

Image is a static version. And **container** is a running version.

For a same image, multiple containers can be running.

Hub.docker.com - is a docker registry which contains more repositories and different versions of different applications. It's a public registry.

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

By default, any container we run is part of bridge network in Docker. Which is an internal network of docker. Unless it is exposed to the host or system where container is running, nobody can access it.

In the above docker command we are making the internal port 5000 to host port 5000.

2. Running container in detached mode.

In detached mode we can run the container in the background without tying the life cycle of container to terminal.

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

#Shows the logs.
docker logs XXXXXX(dockerId)

#Start tailing the logs to keep track.
docker logs -f XXXXXX(dockerId)

#ctrl c to stop tailing the logs.

#running the same image in another container with different port number in detached mode.
docker run -p 5001:5000 -d in28min/todo-rest-api-h2:1.0.0.RELEASE

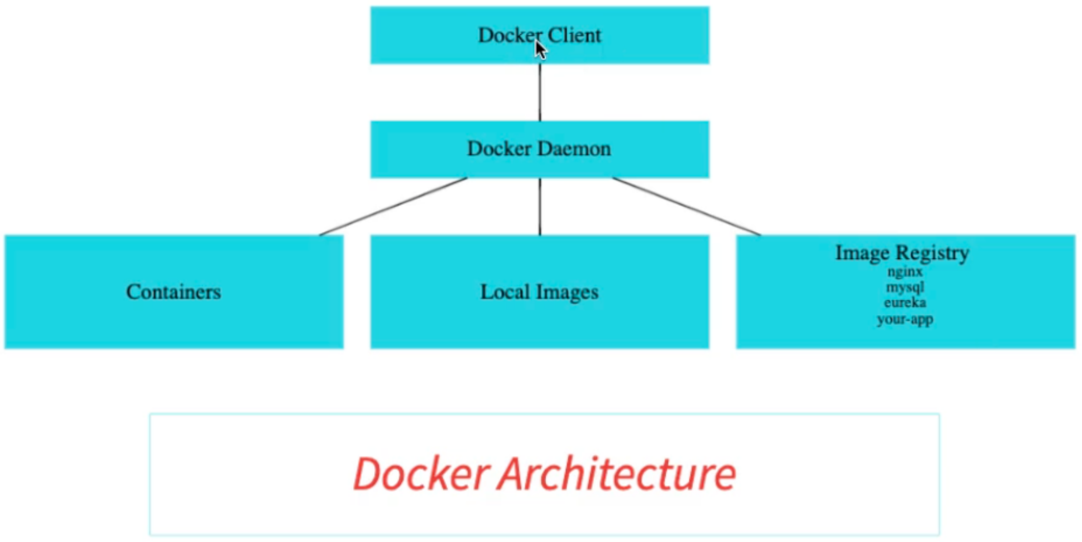
#list the running containers
docker container ls

#list all containers both running and stopped.
docker container ls -a

#list docker images locally
docker images

#stop running container.
 docker container stop id(unique 5 digits not entire id)

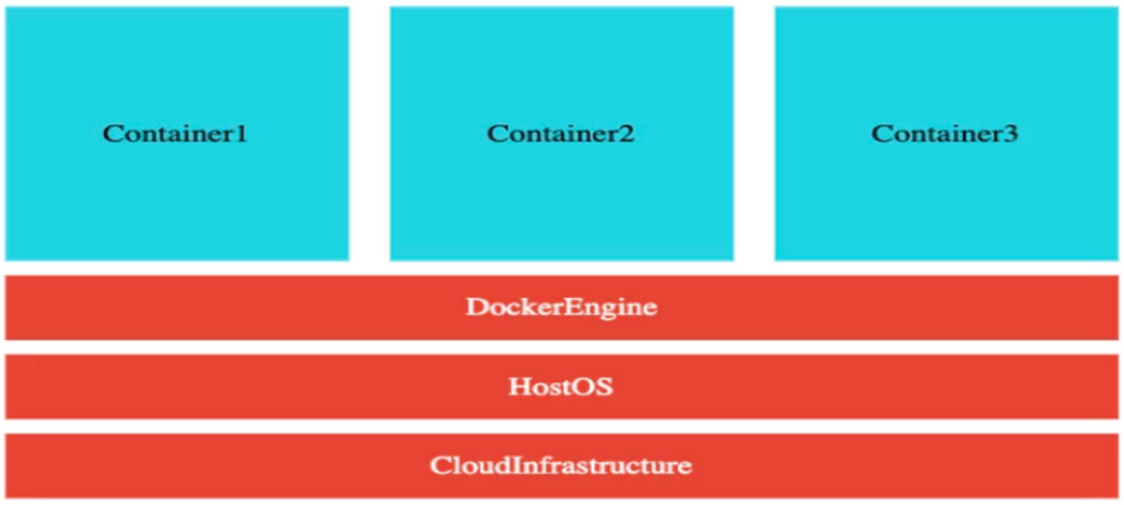
Docker Architecture



All commands done in Docker Client is sent to Docker Daemon.

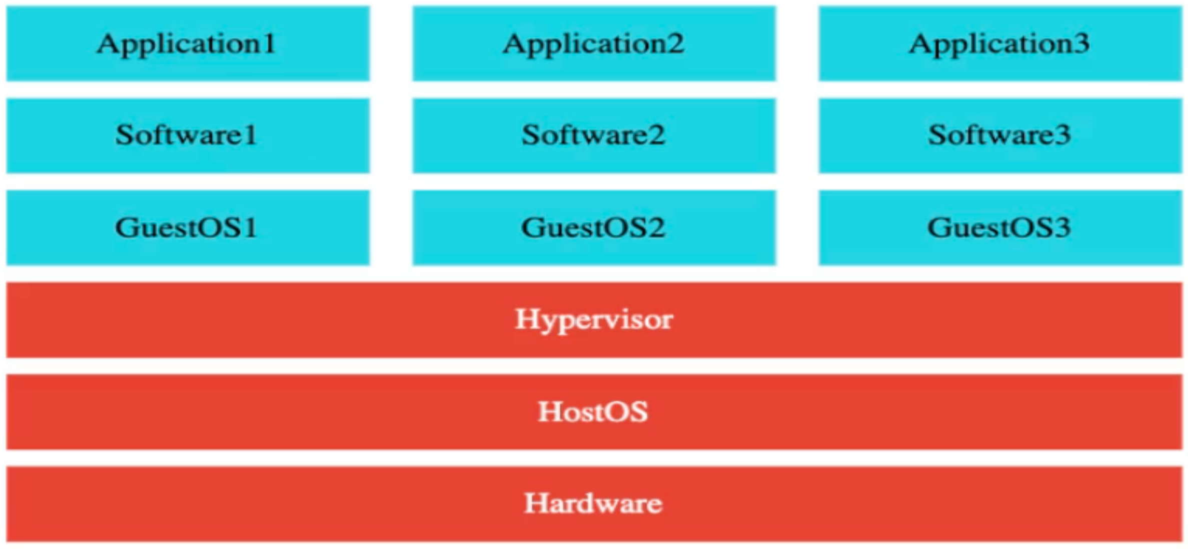
Docker Daemon is responsible for managing containers, Local Images, and pulling, pushing images from registry.

Why Docker?



Deployments using Docker

1. Docker can be installed in cloud.



Deployments using Virtual Machines

2. Before docker VM’s were popular. Hypervisor was managing the Vms.
3. Instead on doing all the VMs and other configurations. We can install docker. Docker takes care of managing all the containers.
4. Docker is lightweight, very efficient.
5. Docker is popular among cloud providers. AWS- Elastic container service, Azure - Azure Container Service

Docker Images:

docker tag in28min/todo-rest-api-h2:1.0.0.RELEASE in28min/todo-rest-api-h2:latest

Single image can have multiple tags.
This will update the image and insert a new image with same image id but different tag.

#mysql is an official image. I.E
docker pull mysql

pull command will download the image to local.

To search an image **docker search mysql**

To check all the steps involved in creating the image:1. docker image history imageId

Docker image inspect imageId - Used to see entry point

#To remove image from local:
docker image remove imageId

Docker Container

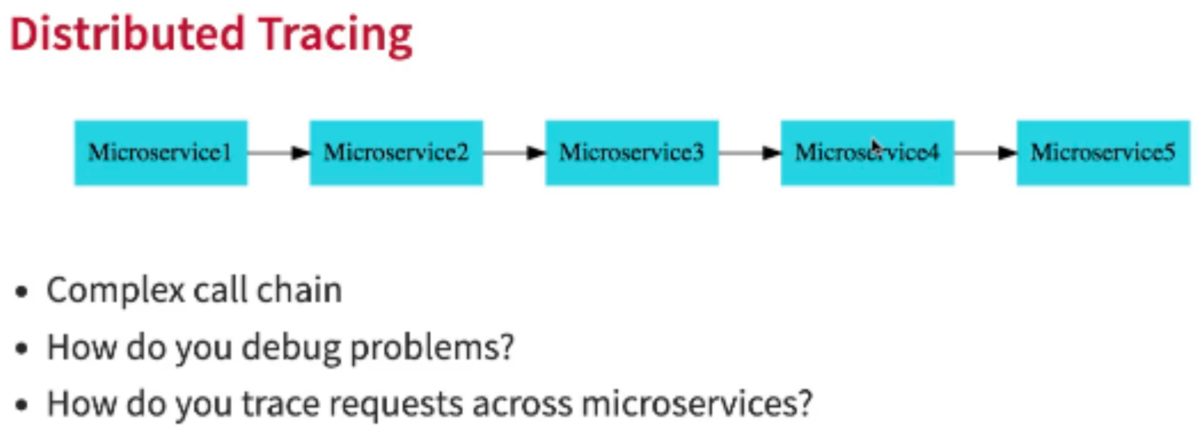
#starting container with restart mode
docker run -p 5000:5000 -d —restart=always in28min/todo-rest-api-h2:1.0.0.RELEASE

While starting the docker container with **restart=always**, whenever the docker daemon is restarted the container automatically starts.

#top process being done in a container
docker top imageId

#df shows images, container, local volumes.
docker system df

Distributed Tracing:

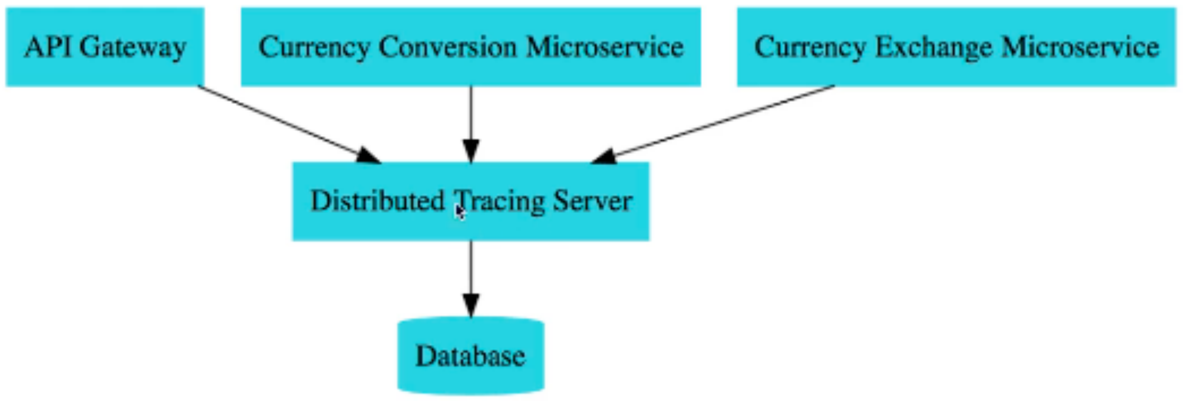


Distributed Tracing

- Complex call chain
- How do you debug problems?
- How do you trace requests across microservices?

Distributed tracing.

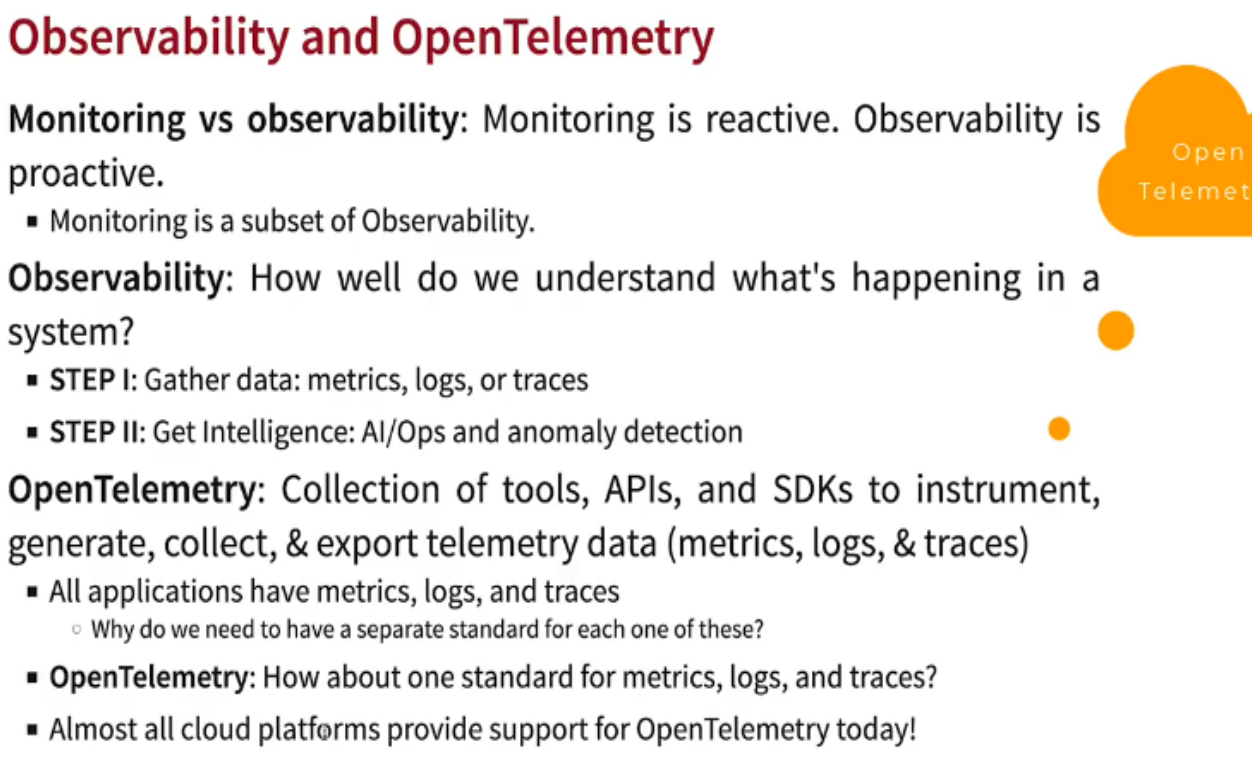
If a request is following throwing multiple microservices and it’s taking a lot of time.
We must find out how much time the request is spending in each microservice and which microservice is consuming is consuming more time.
All the micro-services will send all the information out all tracing information to **distributed Tracing Server (Zipkin)** which will store all the information in Database and it will provide a UI.
And we can query against the UI and find information against the request which are executed.


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

Zipkin Container using Docker

docker run -p 9411:9411 openzipkin.zipkin:2.23

Observability and OpenTelemetry:



Observability and OpenTelemetry

Monitoring vs observability: Monitoring is reactive. Observability is proactive.

- Monitoring is a subset of Observability.

Observability: How well do we understand what's happening in a system?

- **STEP I**: Gather data: metrics, logs, or traces
- **STEP II**: Get Intelligence: AI/Ops and anomaly detection

OpenTelemetry: Collection of tools, APIs, and SDKs to instrument, generate, collect, & export telemetry data (metrics, logs, & traces)

- All applications have metrics, logs, and traces
  - Why do we need to have a separate standard for each one of these?
- **OpenTelemetry**: How about one standard for metrics, logs, and traces?
- Almost all cloud platforms provide support for OpenTelemetry today!

Dependencies:

Adding the dependencies in API Gateway, CurrencyExchangeService, CurrencyConversionService.

```
<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-observation</artifactId>
</dependency>

<!-- OPTION 1: Open Telemetry as Bridge (RECOMMENDED) -->
<!-- Open Telemetry
   - Simplified Observability (metrics, logs, and traces) -->

<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>

<dependency>
   <groupId>io.opentelemetry</groupId>
   <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
#SB3
management.tracing.sampling.probability=1.0
logging.pattern.level=%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]
```

Micrometer will assign an id with specific request. Id will be useful in tracing the request

Creating Docker Images:

Pom.xml

```
<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
				<configuration>
					<image>
						<name>vinothkumarj0006/mmv3-
${project.artifactId}:${project.version}</name>
					</image>
					<pullPolicy>IF_NOT_PRESENT</pullPolicy>
				</configuration>
			</plugin>
		</plugins>
</build>
```

Go to maven build edit/run configurations.

Under run **spring-boot:build-image -DskipTests**

choose the working directory and run the image will start building.

docker run -p 8000:8000 vinothkumarj0006/mmv3-currency-exchange-service:0.0.1-SNAPSHOT

imageId with full path will be generated in the console.

Docker Compose:

Docker compose is a tool for defining and running Muti container docker applications. By configuring a **Yaml** file and by single command we can run all the containers.

docker-compose.yaml

```
version: '3.7'

services:
  currency-exchange:
    image: vinothkumarj0006/mmv3-currency-exchange-service:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports:
      - "8000:8000"
    networks:
      - currency-network
    depends_on:
      - naming-server
    environment:
      EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://naming-server:8761/eureka
      MANAGEMENT.ZIPKIN.TRACING.ENDPOINT: http://zipkin-server:9411/api/v2/spans

  currency-conversion:
    image: vinothkumarj0006/mmv3-currency-conversion-service:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports:
      - "8100:8100"
    networks:
      - currency-network
    depends_on:
      - naming-server
    environment:
      EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://naming-server:8761/eureka
      MANAGEMENT.ZIPKIN.TRACING.ENDPOINT: http://zipkin-server:9411/api/v2/spans

  api-gateway:
    image: vinothkumarj0006/mmv3-api-gateway:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports:
      - "8765:8765"
    networks:
      - currency-network
    depends_on:
      - naming-server
    environment:
      EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://naming-server:8761/eureka
      MANAGEMENT.ZIPKIN.TRACING.ENDPOINT: http://zipkin-server:9411/api/v2/spans

  naming-server:
    image: vinothkumarj0006/mmv3-naming-server:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports:
      - "8761:8761"
    networks:
      - currency-network

  zipkin-server:
    image: openzipkin/zipkin:2.23
    mem_limit: 300m
    ports:
      - "9411:9411"
    networks:
      - currency-network
    restart: always

networks:
  currency-network:
```

Save. And run.

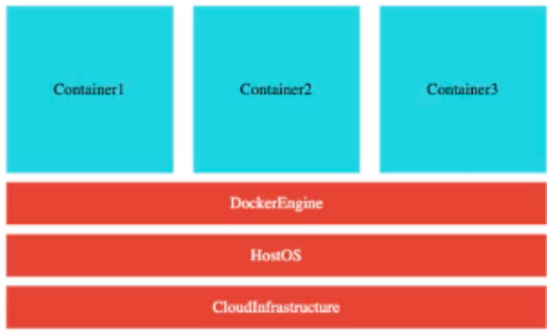
Note: spacing is important in yaml.

#command

```
docker-compose up
```

Docker

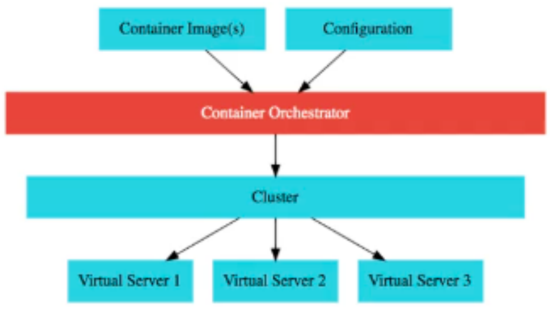
- Create **Docker images** for each microservice
- Docker image **contains everything a microservice needs** to run:
  - Application Runtime (JDK or Python or NodeJS)
  - Application code
  - Dependencies
- You can run these docker containers **the same way** on any infrastructure
  - Your local machine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

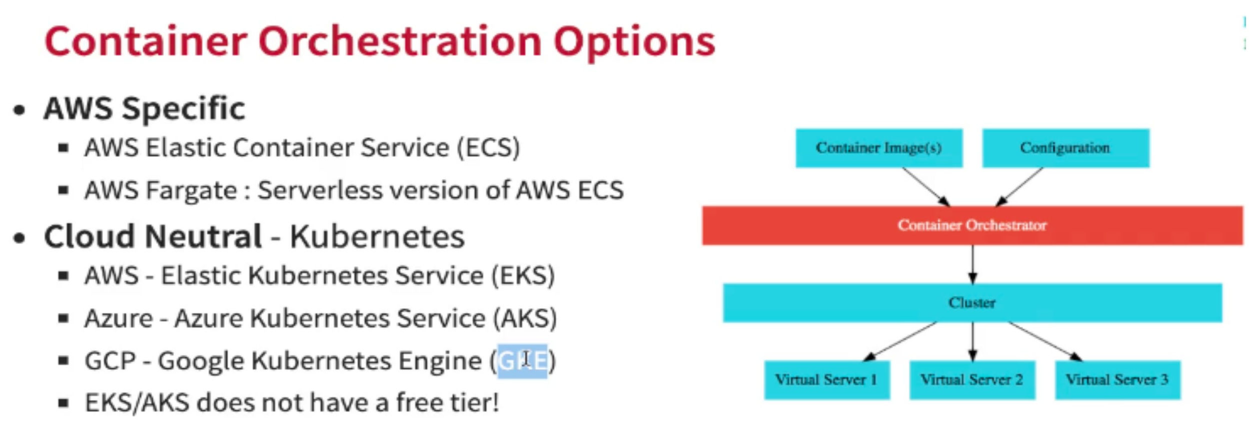


Container Orchestration:

Container Orchestration

- **Requirement** : I want 10 instances of Microservice A container, 15 instances of Microservice B container and ....
- Typical Features:
  - **Auto Scaling** - Scale containers based on demand
  - **Service Discovery** - Help microservices find one another
  - **Load Balancer** - Distribute load among multiple instances of a microservice
  - **Self Healing** - Do health checks and replace failing instances
  - **Zero Downtime Deployments** - Release new versions without downtime





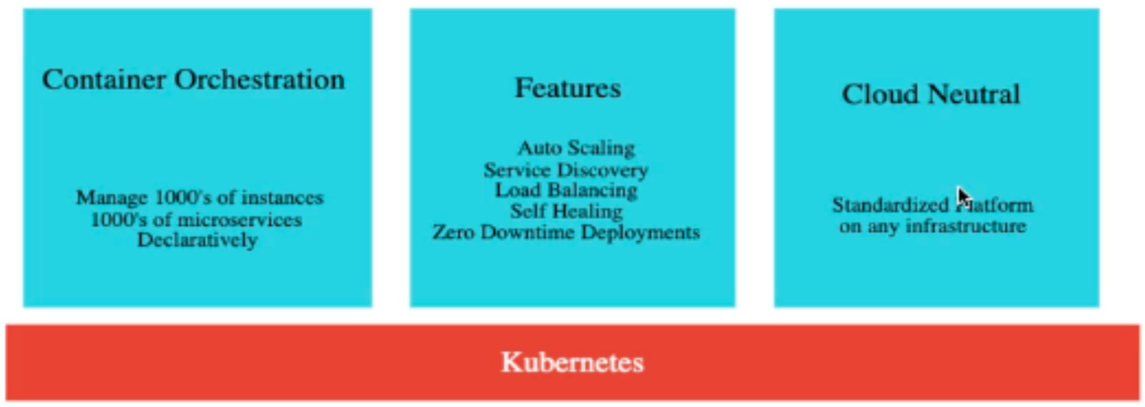
Container Orchestration Options

- **AWS Specific**
  - AWS Elastic Container Service (ECS)
  - AWS Fargate : Serverless version of AWS ECS
- **Cloud Neutral** - Kubernetes
  - AWS - Elastic Kubernetes Service (EKS)
  - Azure - Azure Kubernetes Service (AKS)
  - GCP - Google Kubernetes Engine (GKE)
  - EKS/AKS does not have a free tier!

Kubernetes:

Kubernetes is container orchestration service which helps in managing containers in following processes:

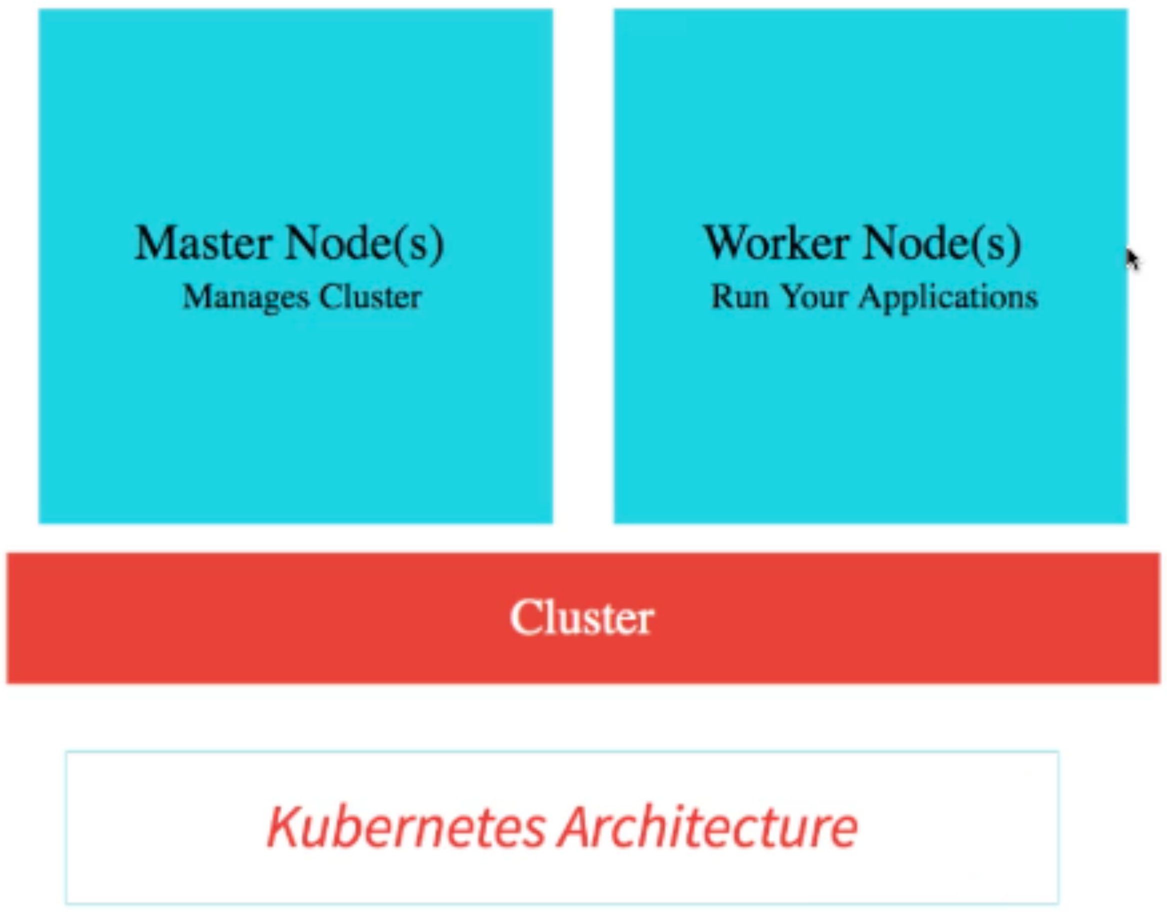
1. Auto Scaling(1000's of microservices can be scaled to 1000's on instance in a declarative way)
2. Service Discovery
3. Load Balancing
4. Monitoring and Self-Healing
5. Zero Downtime DeploymentsKubernetes is Cloud Neutral. AWS, Azure, GCP has different solutions for kubernetes.



Kubernetes Cluster is a resource manager which has combination of nodes and master node. It manages the virtual servers in the cloud.

Virtual servers are called differently among different Cloud servers.
Amazon AWS- EC2
Microsoft Azure - Virtual Machines VMs
Google Cloud - Compute Engine
Kubernetes - Nodes.

Kubernetes on Cloud:
Amazon EKS - Elastic Kubernetes Service
Microsoft AKS - Azure Kubernetes Service
Google GKE - Google Kubernetes Engine



Kubernetes manages 1000’s of Nodes(Virtual machines).

Nodes are managed by Master Nodes. When we need high availability, we need more than one master node. Each master node contains n number of nodes(Worker nodes).

Master Node manages the worker nodes, It ensures the worker nodes are motivated and work is done.

Worker node run the application.

GCP: Kubernetes Engine:

Clusters : Create Cluster and manage them.
Workloads: Manages the application or containers which has to be deployed such as REST APIs, WebApi.
Services & Ingress: It helps to provide access to the external world through theses workloads.
Application Configuration : Configuration to applications.
Storage : persistent storage.

Facts:

Kubernetes - K8S
Koobernetеez pronunciation
Logo - Helmsman

GKS:
Kubernetes Engine -> Create cluster with default config.
Open cmd using connect between.
# command to connect clustergcloud container clusters get-credentials vinothkumarj-cluster --region us-central1 --project wired-victor-419121

# kube cuttel command
 Kubectl

Kubectl - is a kube controller. It will work with any kubernetes cluster irrespective of cluster being in local, cloud or any machine. **Kubectl** used to do all like

1. Deploy an app.
2. Increase no of instances of app.

3. Install new version of the app.**#Deploying an application using Kuberctl to the kubernetes cluster. Image of the rest-api that we want to deploy.**

After successful deployment , deployment id is generated.
kubectl create deployment hello-world-rest-api --image=in28min/hello-world-rest-api:0.0.1.RELEASE

#exposing the deployment to the outside world
kubectl expose deployment hello-world-rest-api --type=LoadBalancer --port=8080

After the deployment is done, in console go to Services and Ingress and check the endpoint URL. Services should be loaded and ready.

How Kubernetes creates and runs an application?

# lists all the events done from the beginning of creating an applciatiion
Kubectl get events
#command to get pods
Kubectl get pods
#get replica set
Kubctl set replicaset

Kubernetes uses single responsibility principle.
Each of pod, service, replica set, deployment an import roles to play. All of them are linked by selectors and labels.

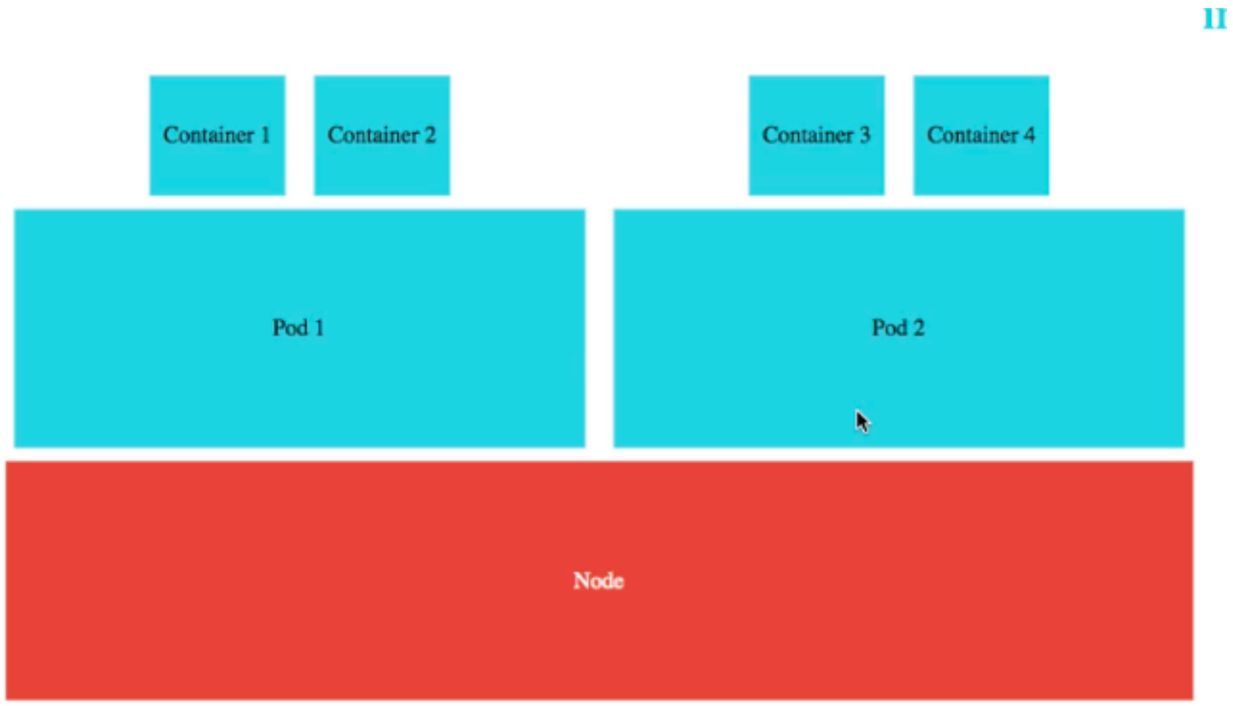
PODS - Wrapper for a set of containers. It has Ip address, labels, annotations…
REPLICA SET- Ensures specific number of pods are always running.
DEPLOYMENT - Ensures a release upgrade happens without a downtime.

When command kubectl create deployment is executed - Kubernetes created deployment, a replica set, and a pod.
When command kubectl create deployment is executed - Kubernetes created a service.

All these have individual responsibilities to play. Such as manage workloads, to enable scaling, to provide external access to workloads and to enable zero time deployment.

PODS: Pod is the smallest deployable unit in kubernetes.

kubectl get pods -o wide



1.   Pod is the smallest deployable unit in kubernetes.
2.   We cannot have a host a container without a pod.
3.   Container lives inside the pod.
4.   Each pod have unique ip address.
5.   A pod can contain multiple containers. All of them share resources. Within a same pod containers can talk to each other with localhost.

6.   A kubernetes node can contain multiple pods and each of the pods can contain multiple containers. These pods can be from different application or of same application.Pod gives a way to put the containers together. It provides IP address and provides categorization with labels.

POD NAMESPACE: Pod namespace provides isolation from parts of the cluster from other parts of the cluster.   By creating different namespace, we can **differentiate it.**

Replica Sets:

Replicaset ensures specific number of pod replicas are running at all times.
#command for replicasets**kubectl get replicasets**
# to run multiple pods in a replica set.**kubectl scale deployment hello-world-rest-api --replicas=3**
kubectl scale deployment hello-world-rest-api --replicas=3

Now the application will run in three pods with multiple application instances.

Deployments:

To update to a new version of an existing and running application and to update it to run a new version of application with zero downtime we use deployments. The strategy is called rolling update. (It updates one pod at a time.)

#kubectl set image deployment name_OF_DEPLOYMENT + name of container + name of the image
kubectl set image deployment hello-world-rest-api hello-world-rest-api=DUMMY_IMAGE:TEST

Even through if we update the existing container with dummy image the application will not go down. It will create new replica set. Replica set will try to launch a pod. But it will fail because the image id is invalid.

When doing current image id:**kubectl set image deployment hello-world-rest-api hello-world-rest-api=in28min/hello-world-rest-api:0.0.2:RELEASE**

Image will be updated.

After releasing a new version v2, the deployment will do:

Create new version of replica set for v2.
1. Replica set will create a pod.
2. Then the deployment will scale up the v2 instances one by one and scale down the v1 instances one by one.
3. Then more pods will be created for v2 instance. And all pods of v1 will be deleted.

Services

#get all services.
kubectl get services

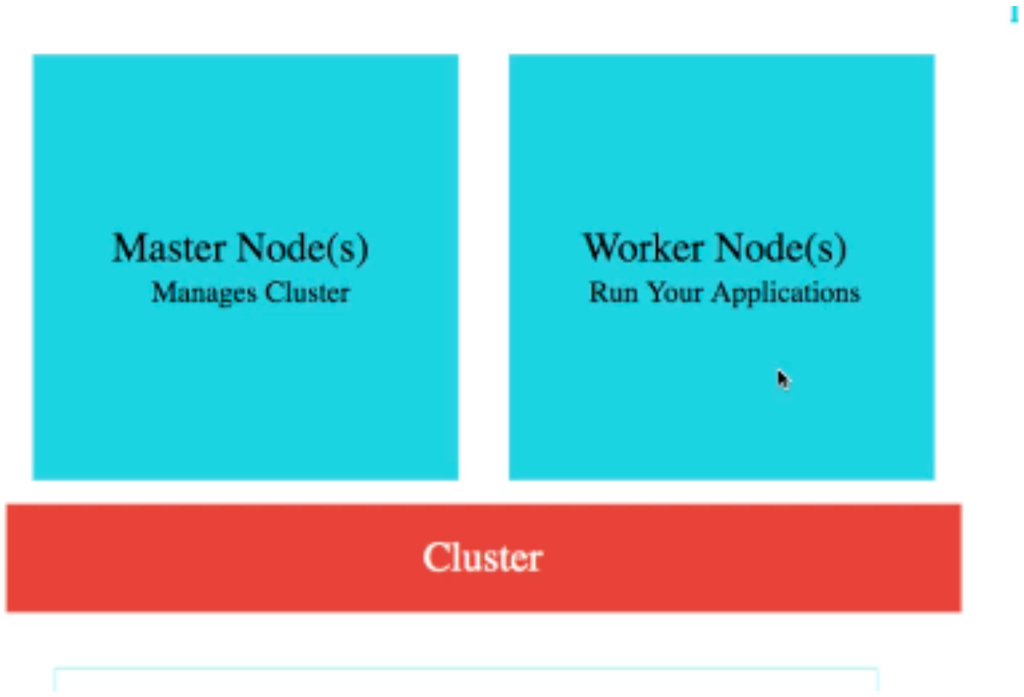
In kubernetes a pod is a throwable unit.

Irrespective of pods being deleted and created we don’t want to change the URL which has pod ip address each time so that the consumer end wont get affected.

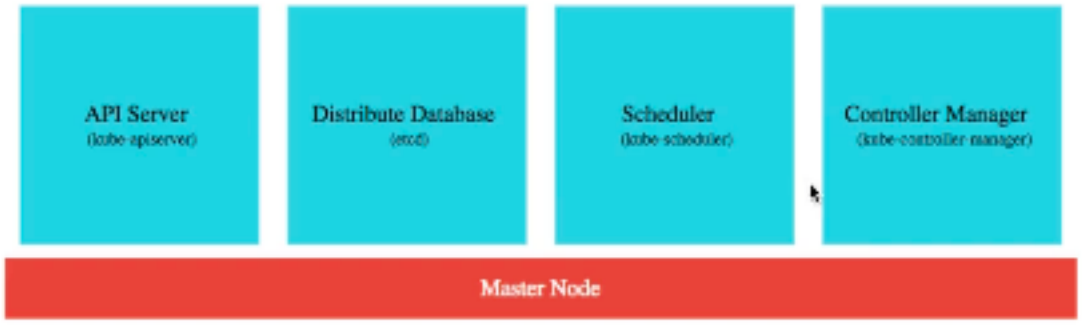
Service provides an always available external external interface to the applications which are running inside the pods. It always to receive tariff through a permeant IP address.

1. Load balancer.
2. Cluster Ip

Kubernetes Architecture:



MASTER NODE:



Kubernetes Architecture

Important part of Master Node:

1. ETCD - Distributed database.

All the configuration changes being done, all deployments being created, scaling details being done are stored in distributed database.

We are setting desired state is stored in distributed database. It is distributed, it is advised to maintain 2 to 3 replicas of distributed database so that the data is not lost.

2. API Server - Kube API Server

If any changes are being done via kubectl or google cloud console the change is submitted to the API server and it is processed through it.

3. Scheduler - Kube Scheduler

Responsible for scheduling the pods onto the nodes based on available CPU ,memory, and port conflicts and considering other factors.

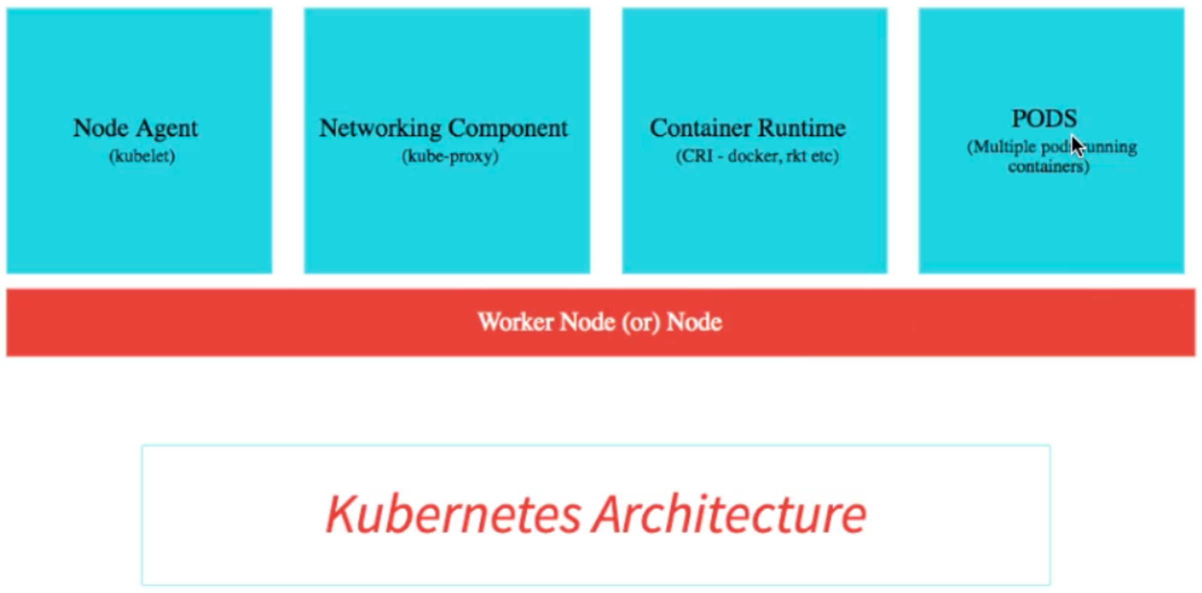
4. Controller Manager - Kube Cuttle Manager

Controls overall health of the cluster. It makes sure the actual state of cluster is matches the desired state. (Like number of instance changes).

Master Node is not responsible for scheduling the user applications.

All the applications will be running in pods in **Worker Node or Node.**

WORKER NODE OR NODE:



1. **Node Agent - Kubelet**

It monitors what's happening on the node and communicates it to master node controller manager.

2. Networking Component - Kube-Proxy

It helps in exposing services around the nodes and pods.

3. Container Runtime - Docker, rkt etc.

It helps to run the container. Most frequently used one is docker. We can use Kubernetes with any OCI(Open container service)

To run the glcoud and kubectl at local cmd we must install them both.

After Installing Gcloud SDK:

Cd downloads/
./google-cloud-sdk/bin/gcloud init

export PATH="$PATH:/Users/vinoth/google-cloud-sdk/bin"

Gcloud init

Install kubectl

kuberctl —version

Connecting the cluster

gcloud container clusters get-credentials vinothkumarj0006 --region us-central1 --project wired-victor-419121

Create docker images in IDE with maven project edit settings.

After Creating the docker image push them into docker registry.

docker push vinothkumarj0006/mmv3-currency-conversion-service:0.0.11-SNAPSHOT

docker push vinothkumarj0006/mmv3-currency-exchange-service:0.0.11-SNAPSHOT

Now create deployment in gcloud IDE.

#creating deployment for currency exchange
kubectl create deployment currency-exchange --image=vinothkumarj0006/mmv3-currency-exchange-service:0.0.11-SNAPSHOT

#exposing it
kubectl expose deployment currency-exchange --type=LoadBalancer --port=8000

Once the services are up.

#ip is external endpoints.
curl http://34.68.12.146:8000/currency-exchange/from/USD/to/INR
#output
{"id":10001,"from":"USD","to":"INR","conversionMultiple":65.00,"environment":"8000 v11 currency-exchange-7cc5957fcb-h62dj"}

#creating deployment for currency conversion
kubectl create deployment currency-conversion --image=vinothkumarj0006/mmv3-currency-conversion-service:0.0.11-SNAPSHOT

#exposing it
kubectl expose deployment currency-conversion --type=LoadBalancer --port=8100

#ip is external endpoints http://34.29.196.32:8100/
curl http://34.29.196.32:8100/currency-conversion-feign/from/USD/to/INR/quantity/10

Whenever new service starts up, new environment variables are created automatically will all the available service with a specific pod by kubernetes for service discovery.

#service name + service_port all pods..

CURRENCY_EXCHANGE_SERVICE_PORT
CURRENCY_CONVERSION_SERVICE_PORT

As the services are working, Now we can see how to configure the configurations manually(Declarative approach.).

cd /Users/vinoth/spring-microservices-v3-main/05.kubernetes/currency-exchange-service

#checking the current yaml
kubectl get deployment currency-exchange -o yaml

#saving the current **deployment.yaml** to do the changes locally.
kubectl get deployment currency-exchange -o yaml >> deployment.yaml

#saving the current **service.yaml** to do the changes locally.
kubectl get service currency-exchange -o yaml >> service.yaml

Then try to save both yaml in a single yaml file by separating it with - - -
And save it. After changing the number of replicas from 1 go 2

#shows the changes doen in saved deployments.yaml
kubectl diff -f deployment.yaml

#
kubectl apply -f deployment.yaml

In deployment.yaml we are specifying the desired sates, and kubernetes is comparing the desired state with active state and does the changes this is called Declarative approach.

Service Discovery and Load Balancing done by Kubernetes for free.

Deployments.yaml: Let’s explore necessary things:Let’s say we have a pod with container running.

```
apiVersion: apps/v1 # definiton of deployment
kind: Deployment
metadata:
  annotations:
    autopilot.gke.io/resource-adjustment: '{"input":{"containers":[{"name":"mmv3-currency-exchange-
service"}]},"output":{"containers":[{"limits":{"cpu":"500m","ephemeral-
storage":"1Gi","memory":"2Gi"},"requests":{"cpu":"500m","ephemeral-
storage":"1Gi","memory":"2Gi"},"name":"mmv3-currency-exchange-service"}]},"modified":true}'
    autopilot.gke.io/warden-version: 2.7.52
    deployment.kubernetes.io/revision: "1"
  labels: # deployment name and namespace details.
    app: currency-exchange
  name: currency-exchange
  namespace: default
spec: # we try and match deployment to the pod with matchlables
  replicas: 2 # number of pods needed
  selector:
    matchLabels:
      app: currency-exchange # this deployment will be matching for all the pods with the specified
name.
  strategy: # how an update to the deployment is done.
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate #stragetegy type
  template:
    metadata:
      labels: # to the entire pod the name is given in table
        app: currency-exchange
    spec:# specification of the container below.
      containers: # we can have multiple containers in a pod
      - image: vinothkumarj0006/mmv3-currency-exchange-service:0.0.11-SNAPSHOT #used to
create containers
        imagePullPolicy: IfNotPresent #values can be always or ifnotpresent
        name: mmv3-currency-exchange-service
      restartPolicy: Always #policy
---
apiVersion: v1 # definiton of service
kind: Service
metadata:
  annotations:
    cloud.google.com/neg: '{"ingress":true}'
  labels:
    app: currency-exchange # name definded to the service
  name: currency-exchange # name attached to the service
  namespace: default. # used to define and differentiate the prod,dev,uat environments.
spec: #specification of the service matches to deployment.
  ports: #expose ports which match the deployment name mentioned in selector
    - port: 8000
      protocol: TCP
      targetPort: 8000
  selector:
    app: currency-exchange
  sessionAffinity: None # sending requests to same service
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 34.68.12.146
```

Enabling Logging and Tracing APIs in GCP

GCP -> API AND SERVICES - ENABLING APID AND SERCICES -> + CLOUD LOGGIN APIU

Deleting all the existing pods and deployment the manually created yaml file.

# deleting all the existing pods of currency exchange
kubectl delete all -l app=currency-exchange

# deleting all the existing pods of currency conversion
kubectl delete all -l app=currency-conversion

#deploying the new yaml file
Kubectl apply -f deployment.yaml

#hitting the service with newly created external ip for currency-exchange
curl http://35.226.1.125:8000/currency-exchange/from/USD/to/INR

#hitting the service with newly created external ip for currency-conversion
curl http://34.16.2.69**:8100/currency-conversion-feign/from/USD/to/INR/quantity/10**

declarative approach done:

#NOTE:
Whenever new service starts up, new environment variables are created automatically will all the available service with a specific pod by kubernetes for service discovery.

CURRENCY_EXCHANGE_SERVICE_PORT
CURRENCY_CONVERSION_SERVICE_PORT

The problem with this approach is when starting up the currency conversion at that point of time when currency exchange service is down or not available then currency conversion service will not get the environment variables of currency exchange service.

Because only live services URL's will be provided to the currency conversion service.

Which is why customer environment variables is preferred over default environment variables.

CURRENCY_EXCHANGE_URI

```
#in currency-conversion-service
@FeignClient(name = "currency-exchange", url =
"${CURRENCY_EXCHANGE_URI:http://localhost}:8000")
```

After changing the pom version and the property in proxy class, create new docker images and push it to docker hub.

```
docker push vinoth3-currency-conversion-service:0.0.12-SNAPSHOT
docker push vinothkumarj0006/mmv3-currency-exchange-service:0.0.12-SNAPSHOT
```

Then change the deployments.yaml image name and add environmental value as below on the container.

```
containers:
- image: vinothkumarj0006/mmv3-currency-conversion-service:0.0.12-SNAPSHOT
  imagePullPolicy: IfNotPresent
  name: mmv3-currency-conversion-service
  env:
    - name: CURRENCY_EXCHANGE_URI
      value: http://currency-exchange
restartPolicy: Always
```

By mentioning it in the yaml instead of using the default env variables helps us to look for all currency-exchange services whenever load balancing loads.

```
#check differences in yaml
Kubectl diff -f deployment.yaml
```

#deploy differences in yaml

kubectl apply -f deployment.yaml

Centralized Configuration Of Environment Variables (ConfigMaps acts as centralized configuration in Kubernetes)

Instead of hardcoding the env variables in yaml be can do centralized configuration.

```
kubectl create configmap currency-conversion --from-literal=CURRENCY_EXCHANGE_URI=http://currency-exchange

kubectl get configmap currency-conversion

kubectl get configmap currency-conversion -o yaml
```

```
apiVersion: v1
data:
  CURRENCY_EXCHANGE_URI: http://currency-exchange
kind: ConfigMap
metadata:
  creationTimestamp: "2024-04-05T00:37:07Z"
  name: currency-conversion
  namespace: default
  resourceVersion: "1827517"
  uid: 85fbe31f-bf95-46ad-bb55-16abc6a99b79
```

We can now add these changes in deployment.yaml(Deployment details, pods details, service details, **+ newly adding** configuration details)

Now changing the yaml container configuration to take env from centralized configuration.

```
containers:
- image: vinothkumarj0006/mmv3-currency-conversion-service:0.0.12-SNAPSHOT
  imagePullPolicy: IfNotPresent
  name: mmv3-currency-conversion-service
  envFrom:
  - configMapRef:
      name: currency-conversion
```

save this in **deployment.yaml**

```
#check differences in yaml
Kubectl diff -f deployment.yaml

#deploy differences in yaml
kubectl apply -f deployment.yaml
```

Checking Logs and Monitoring in GCP

KUBERNETES CLUSTER -> vinothkumarj0006 -> Features -> **Logging** -> view logs -> chose container -> view logs.

KUBERNETES CLUSTER -> vinothkumarj0006 -> Features -> Cloud Monitoring -> **view GKE Dashboard(Shows all metrics)** -> Namespaces, its, nodes, workloads, Services, pods, containers.

Microservices Deployments in Kubernetes

```
# to check overall history
kubectl rollout history deployment currency-conversion
```

1. Kubernetes while deploying the new version if it's failing the old version will keep on be running so that the users won't get impacted.

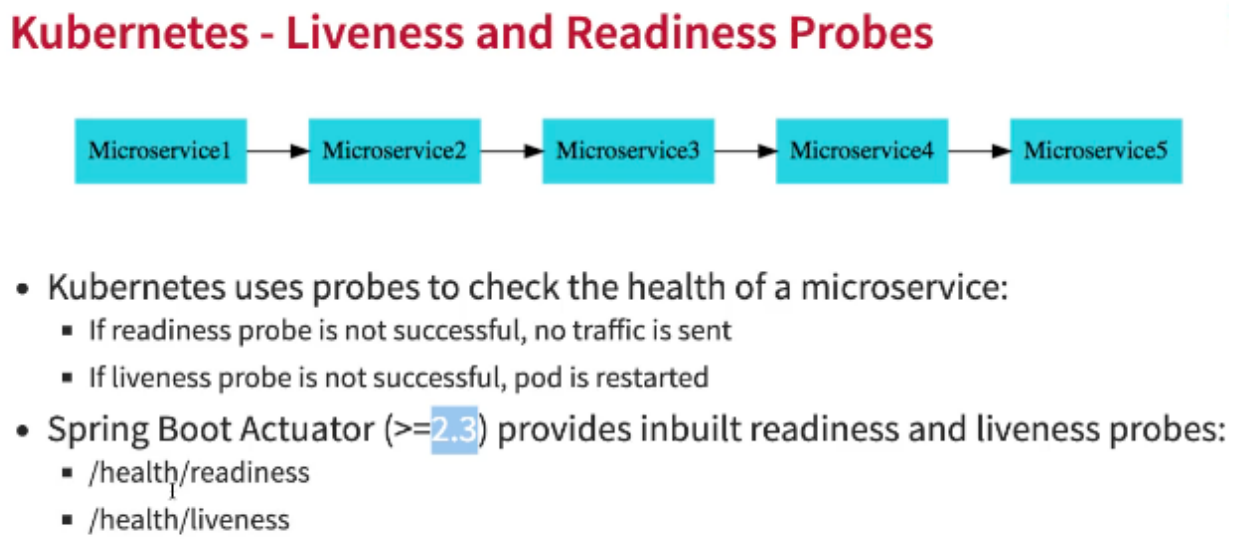
#rollout the newly deployed version

```
Kubectl rollout under deployment currency-conversion - -to-revision=1
```

Configuring Liveness and Readiness

While doing a deployment to make it with zero downtime by using liveness and readiness probes.

Liveness and readiness probes helps us to maintain the application error free, it prevents pods deletion before it can recieve the traffic.



Kubernetes - Liveness and Readiness Probes

Microservice1 → Microservice2 → Microservice3 → Microservice4 → Microservice5

- Kubernetes uses probes to check the health of a microservice:
  - If readiness probe is not successful, no traffic is sent
  - If liveness probe is not successful, pod is restarted
- Spring Boot Actuator (>=2.3) provides inbuilt readiness and liveness probes:
  - /health/readiness
  - /health/liveness

In the deployments.yaml we have adding the readiness url and liveness url under container:

```
containers:
- image: vinothkumarj0006/mmv3-currency-exchange-service:0.0.12-
SNAPSHOT
  imagePullPolicy: IfNotPresent
  name: mmv3-currency-exchange-service
  readinessProbe:
    httpGet:

      port: 8000
      path: /actuator/health/readiness
  livenessProbe:
    httpGet:
      port: 8000
      path: /actuator/heath/liveness
restartPolicy: Always
```

Autoscaling Microservices in Kubernetes

#send 10 request every second to check system performance:
watch -n 0.1 curl http://34.16.2.69:8100/currency-conversion-feign/from/USD/to/INR/quantity/10

#enabling horizontal pod scaling with respect to cpu utilization percentage
kubectl autoscale deployment currency-exchange - -min=1 - -max=3 - -cpu-percent=5

#to check **horizontal pod scaling(spa)**
Kubectl get hpa

#deleting horizontal pod scaling
kubectl delete hpa currency-exchange