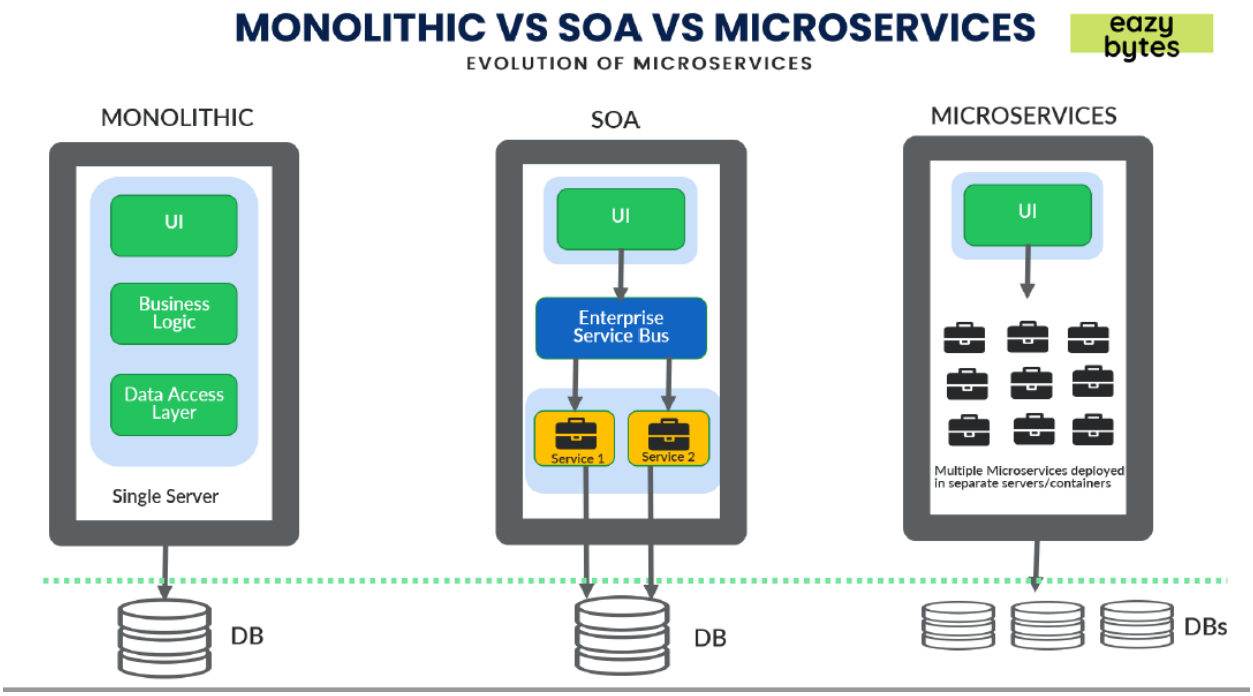
Evolution of Microservices

Before advent of Microservice, there were other flavors of architecture: Monolithic, SOA

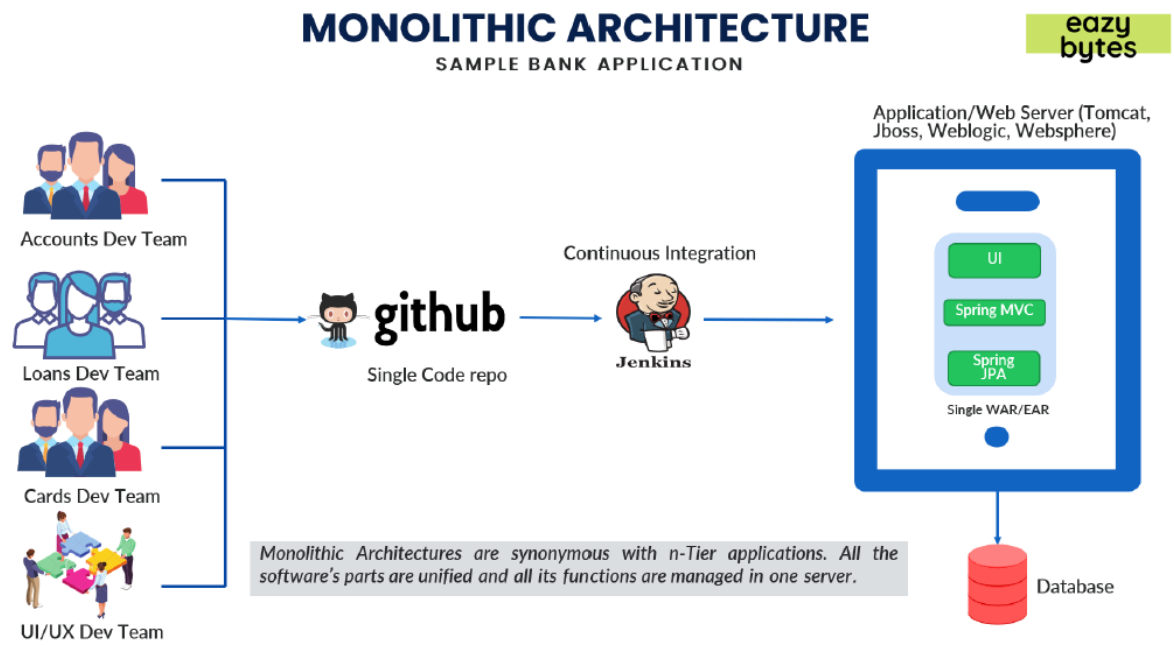


a) In Monolithic, everything from UI, Business Logic, DAO is deployed in one single server. The artifact is a single war or ear file using typically a single Java project. If you want to increase horizontal scalability, then entire set up needs to be copied and deployed over other servers with a Load Balancer in front.

b) In SOA based architecture, UI and Service Layer are in different servers and they interact via an ESB. Typically, this is a SOAP based communication. With SOAP we have additional complexity involved like message size and structure.

c) In case of Microservice based architecture we have different microservice for each component, typically deployed in a container and orchestrated through an orchestration serviced like K8s. In this case scaling up and scaling down of an individual component is easy. Moreover, each Microservice can have its own database.

Monolithic Architecture



a) A change in one module, say Loans will require deployment and testing of entire application.

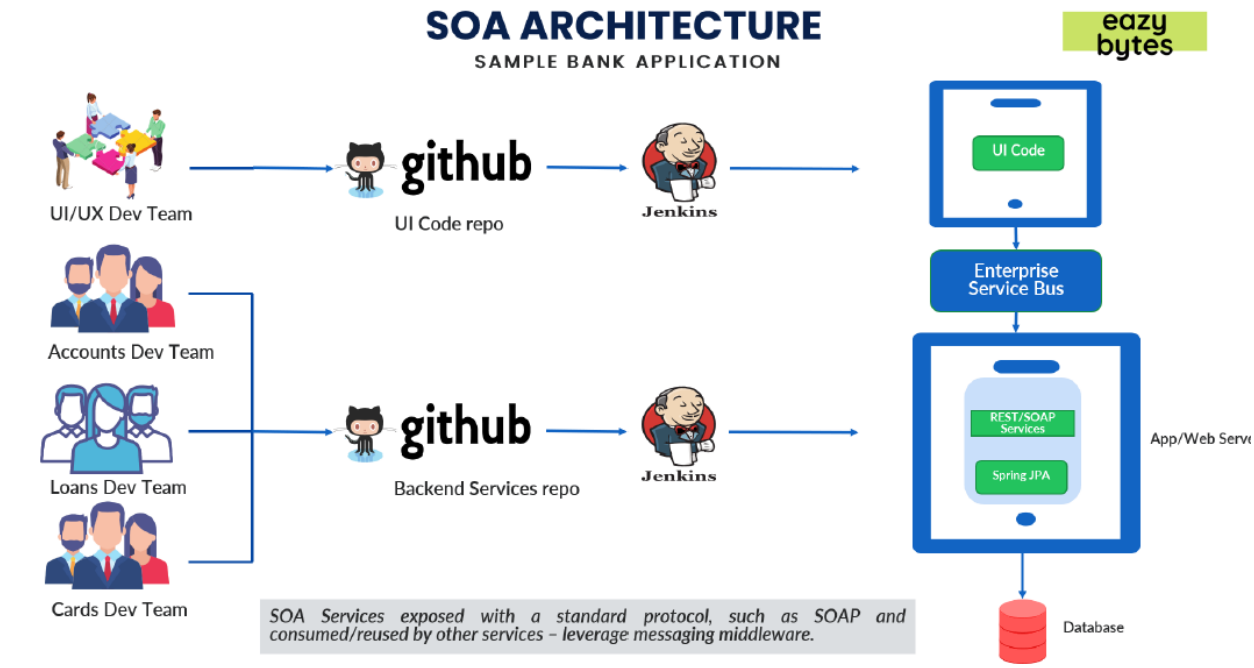
Pros

- Simpler development and deployment for smaller teams and applications
- Fewer cross-cutting concerns
- Better performance due to no network latency

Cons

- Difficult to adopt new technologies
- Limited agility
- Single code base and difficult to maintain
- Not Fault tolerance
- Tiny update and feature development always need a full deployment

SOA Architecture



a) UI and Service Layer separated
b) UI and Service Backend development can be done separately
c) But communication between UI and Service Layer is via ESB (to take care of Security and other cross cutting concerns)
d) But still services cannot be maintained separately
e) SOAP based service suffers from schema definition and message size limitations

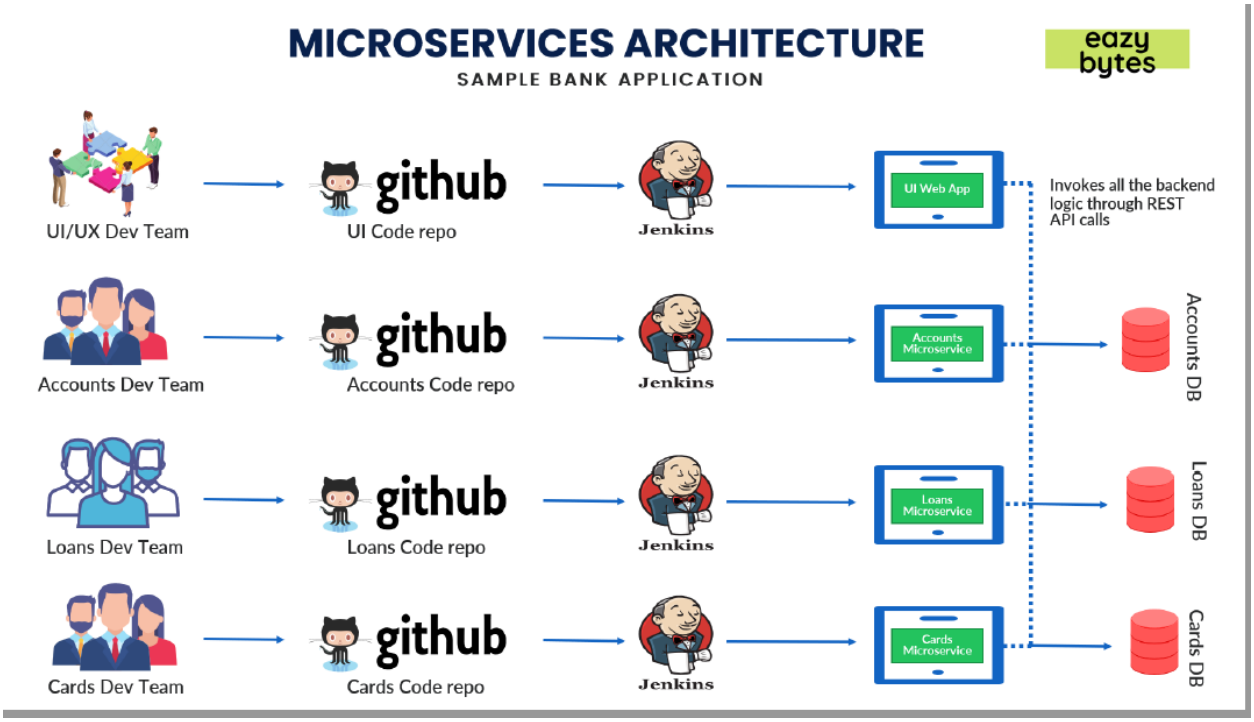
Pros

- Reusability of services
- Better maintainability
- Higher reliability
- Parallel development

Cons

- Complex management
- High investment costs
- Extra overload

Microservices Architecture

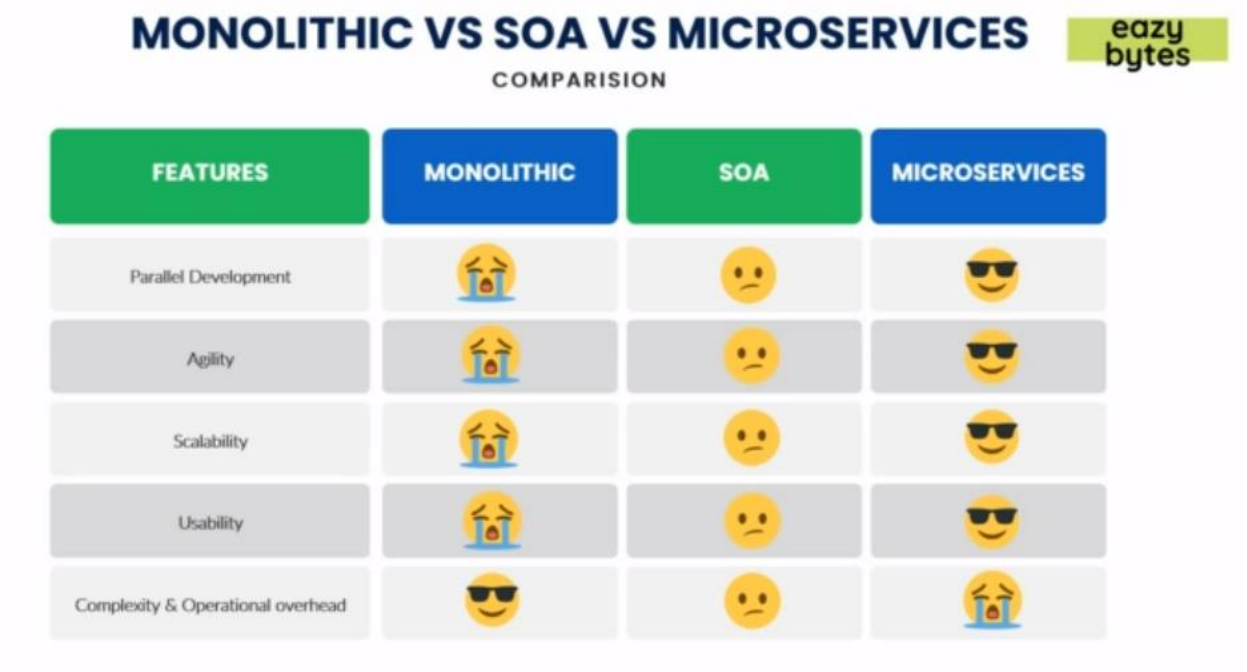


Pros

- Easy to develop, test, and deploy
- Increased agility
- Ability to scale horizontally
- Parallel development

Cons

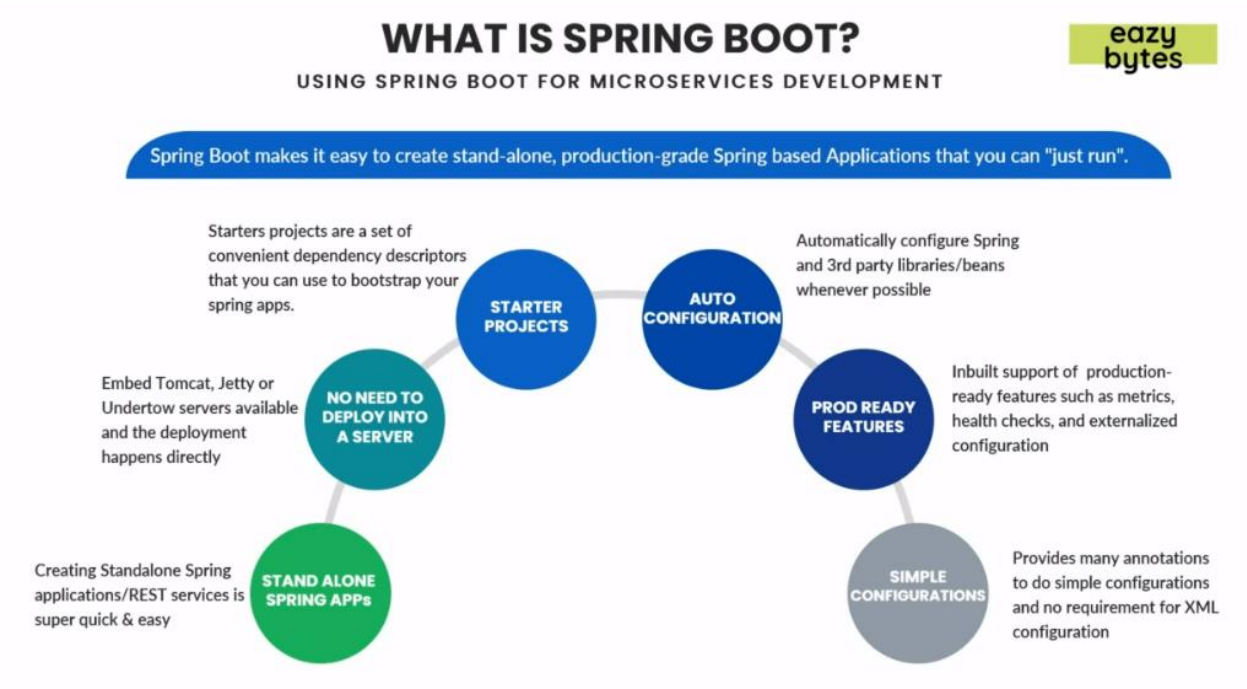
- Complexity
- Infrastructure overhead
- Security concerns



WHY SPRING FOR MICROSERVICES

a) Spring supports modular architecture and loose coupled modules which in principle, philosophy of Microservices based architecture
b) Spring Boot makes development of Microservices fairly very simple.
c) Spring Cloud helps in overcoming challenges associated with Microservices development
d) Provides production ready features like metrics, security, embedded servers
e) Spring Cloud makes deployment of Microservices to cloud very easy

WHAT IS SPRING BOOT



- When you add dependency of Spring Actuator in your project then you will have an added feature of monitoring metrics and health checks by simply hitting a url.
- Spring starter project helps in building Spring Boot projects easy where automatically pom.xml and dependencies will be created

CREATING A SAMPLE MICROSERVICE USING SPRING STARTER PROJECTS

a) Login to spring.initializer
b) Choose your build and programming language
c) Add any dependency like Web, Actuator etc
d) Download project zip, unzip and import it in eclipse
e) Class annotated with SpringBootApplication is main class
f) When you execute the jar, the main method of the class with annotation SpringBootApplication will be called. The main method has code:
SpringApplication.run(HelloSvcApplication.class, args

Run method will if there is any class annotated with Web, so in this case Spring will initialize Spring Web Application Context. It will then wire up all dependencies automatically.

g) After running main class you will see in logs that app will be up in port 8080 by default. If you want to change the port then provide given below entry in application.properties file:
server.port=8082

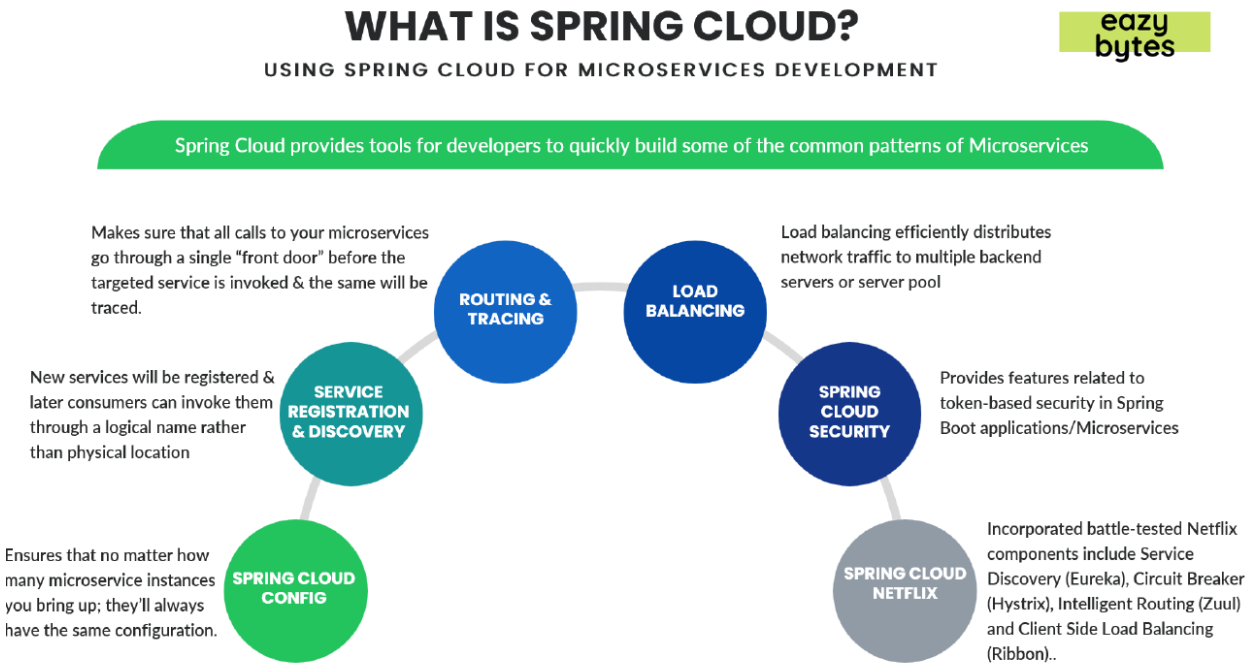
h) Since we added Spring Actuator dependency, hence various metrics and monitoring stats will be available for the Microservice. Access url:
Localhost:8080/actuator

- If you access url http://localhost:8082/actuator/health, then it will show whether service is up or not
- By default not all metrics are available (health and info). To get detail about more metrics add below entry in application.properties file :
management.endpoints.web.exposure.include=*
and then restart the application

i) Hit the url: http://localhost:8082/actuator once again. This time lots of metrics will be available. For example, if you hit url:
http://localhost:8082/actuator/metrics/process.uptime
then it will show for how much time service is up

USING SPRING CLOUD FOR MICROSERVICES DEVELOPMENT

Spring Cloud goes a step ahead and provides various features which makes Micro Services Development fast and easy. Some of the features of Spring Cloud Project are given below:



a) Spring Cloud Config: Used in externalization (outside the repo or source code) of properties like DB Details, Email Details etc. All instances of microservice will use the same configuration.
b) Service Registration and Discovery: Used for discovering and calling Microservice. Suppose you have a microservice which is deployed in a container and you call it from UI using the entire url. What if container instance goes down and a new instance in launched, now ip address and port may change, hence the old url called from UI cannot be used. Therefore we use Service Registration and Discovery to call Microservice by a logical name rather than via physical location. Service Discovery will take care of routing call to actual end point.
c) Routing and tracing: Ensure that call to all microservices goes through a common front door (Service Registry and Discovery) and also tracing the entire route of the call. For example a request can span calls to 10 Microservices. Tracing will help to trace calls to all microservices which can help in identifying any failure point.
d) Load Balancing: Ensure that at any given point request is event distributed among all instances
e) Spring Cloud Security: Microservices are running in container. This feature enables that call to each microservice is secured so that everyone cannot make call to your microservice.
f) Spring Cloud Netflix: Netflix provided its own researched and developed components like Eureka, Hystrix, Zuul and Ribbon to Spring Cloud Community as Open Source

SIZING MICROSERVICES AND IDENTIFYING BOUNDARIES

When you want to design a new Microservices based architecture or migrate an existing monolithic then we need to make sure that Microservice is not very big (system will not be modularized) or too small (too many microservices which can introduce latency)

For these two approaches are followed in general:

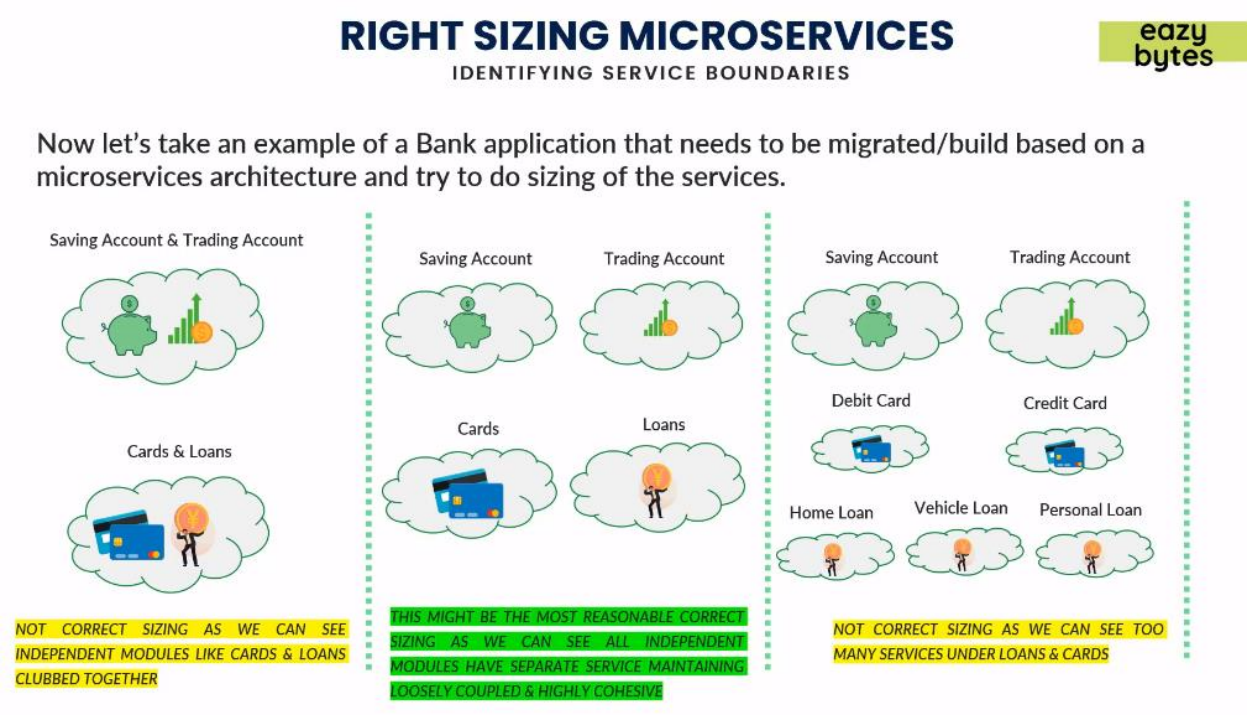
a) Domain Driven Sizing: Deciding boundaries based on business domain and components:
- Needs discussion with leaders who should have very good knowledge of business domain
- For example, in a bank application we can have Accounts department, loans department, stocks department. So, we can have a microservice for each business component
- This process usually takes lots of time (maybe months), as we need to make sure that microservice is too big and not too small and requires a thorough discussion with business leaders.

b) Event Storming Sizing: Deciding boundaries based on events:
- This approach can be followed when we do not have luxury if time and we do not have business experts
- In this case you conduct meetings (with POs, QA and client) to know what are all possible events which can occur in the product. What triggers the event and wants the reaction to event?

- For example, there can be Payment Completed event. Trigger for this event can be when customer selects an item, reaction will be shipping the item.

  But there is no right or prefect approach, you will evolve in sizing in due course of product development.

Example:



Example of migrating an existing Monolithic based e-commerce start up:

MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy bytes

Problem that E-Commerce team is facing due to traditional monolithic design

Initial Days

- It is straightforward to build, test, deploy, troubleshoot and scale during the launch and when the team size is less

Later after few days the app/site is a super hit and started evolving a lot. Now team has below problems,

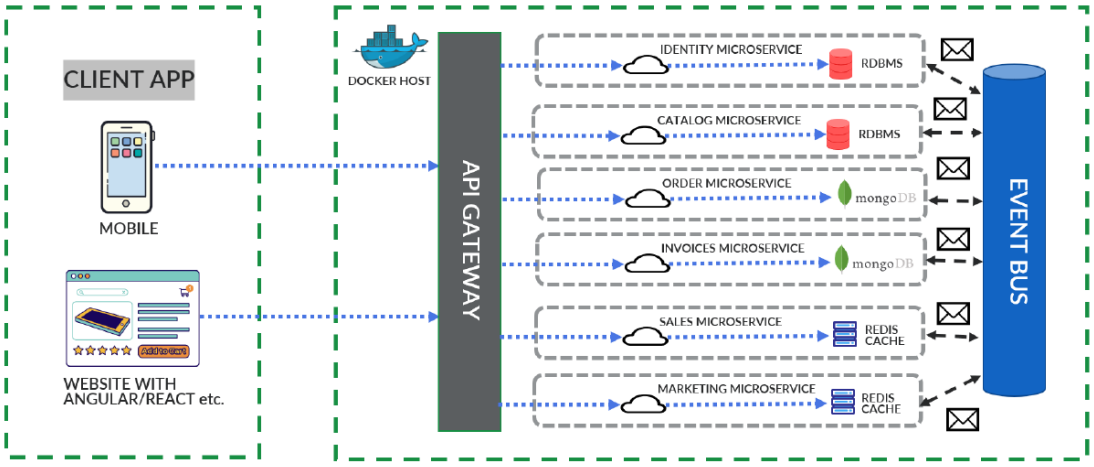
- The app has become so overwhelmingly complicated that no single person understands it.
- You fear making changes - each change has unintended and costly side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release as small as possible and requires a full deployment of the entire application.
- One unstable component can crash the entire system.
- New technologies and frameworks aren't an option.
- It's difficult to maintain small isolated teams and implement agile delivery methodologies.

MONOLOTHIC TO MICROSERVICES

MIGRATION USECASE

eazy bytes

So the Ecommerce company decided and adopted the below cloud-native design by leveraging Microservices architecture to make their life easy and less risk with the continuous changes.



- Microservice for each component
- From UI or app, call will be diverted via API Gateway. Gateway will decide where to route the request.
- Each Microservice has its own db, based on its nature and requirement.
- Asynchronous communication is handled via a Event Bus, say Kafka.

CREATING MICROSERVICES USING SPRNG I/O STARTER

1) While creating project in Spring Starter, choose dependencies as Web, Actuator, H2DB and lombok
2) H2DB is an internal In Memory Data Base provided by spring. It's used for POC Purpose. When you start your application then Spring Boot Framework will automatically execute data.sql file under resources folder. Its will create table and insert data as per scripts provided in data.sql file. When you stop the application all tables and data will be deleted.
3) Project Lombok will generate Getter and Setter automatically for you with proper annotation, but you have to execute the downloaded jar and provide the eclipse location during execution of Lombok jar
4) When you start the spring boot application, you will get message like:
   H2 console available at 'h2-console'. Database available at 'jdbc:h2:mem:testdb'
   Access url localhost:<port where web app is deployed>/h2-console

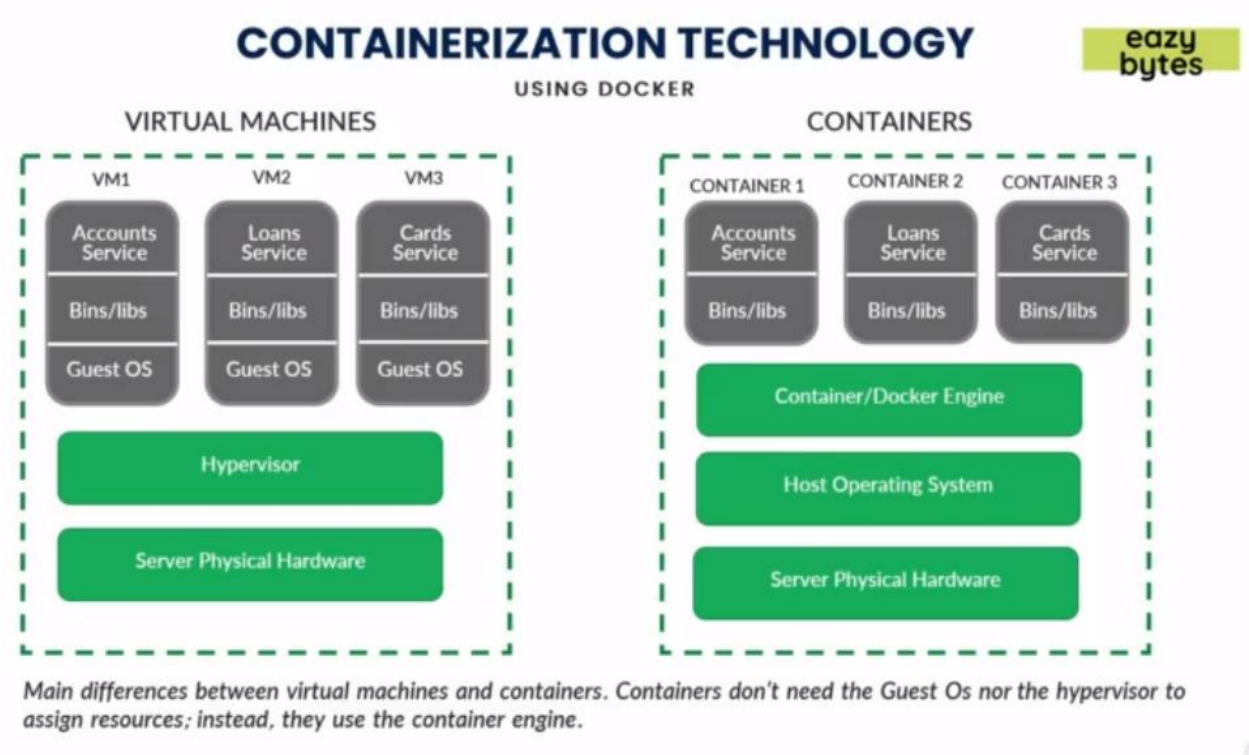
   In the rendered screen provide jdbc:h2:mem:testdb' for JDBC url. Here you will be able to see the In memory DB.

5) Please note that in model class, class name should exactly match the corresponding table name or if that's not the case then explicitly provide table name using annotation @Table.
   For example,
   @Entity
   @Table (name="accounts")
   Public class Account{

   }

DOCKERIZING MICROSERVICES

1) Docker or Containerization solves problems related to deployment, scalability, portability of Microservices
2) Containerization technology:



Main differences between virtual machines and containers. Containers don't need the Guest Os nor the hypervisor to assign resources; instead, they use the container engine.

- Launching a separate VM for each MS will be very expensive and time taking
- VM based approach in general takes time as each VM has a Guest OS which will always take time to boot when VM is restarted
- Container do not have a Guest OS; their boundaries and space are limited. Each container will only occupy space that is required to run, and it will only contain libs related to that Container. For example, Accounts Service container can be Java 11 based, Loans on Java 14 and Cards on Python.
- Adding, removing, and starting containers is also very fast.
- In a container a software package run in isolation. The software package is set of dependent libraries and code required to run the software.
- Software containerization is a technology that is used to deploy and run containers without VMs
- Creating Docker containers out of Docker images is like creating instances of Java class
- An image can have multiple number of containers depending on load
- The way Docker images are built makes it env/platform compatible. The docker image which is used in developer env can be used in test env or in cloud.

kYou build docker image using:

```
docker build . -t <repo-name>
```

repo- name is important.
Repo-name is of format, <docker user name/ repository name in docker hub>

Suppose your docker hub user id is hitesh791. In your docker hub account you have a repository with name accounts, then create docker image using command:
docker build . -t hitesh791/accounts

Above command will create a docker image with name (or repository name) as hitesh791/accounts with tag as latest.

Now then to push the image to docker hub use command: docker push hitesh791/accounts:latest.

These are the step you must follow to build and push image to docker hub

When we create a docker file then we pull open JDK image using
FROM openjdk:8
So here openjdk is repository name when open jdk image was pushed to docker hub.

If we do not specify, tag name then docker will generate a random number to tag your image

- Suppose your docker image id is 1234
  Docker container run -p 8080:8080 -itd 1234
  Or you can also use docker image repo name instead of image id

This will launch a new container where its port 8080 will be exposed as 8080 in outside world

If we want to create another container instance then we can use the same port for docker container as each container will have its own file system, port and network but to outside world we have to change the port as 8080 port is already used:

Docker container run -p **8081**:8080 -itd 1234

Now we have two container instances running for the image with id 1234

3) Docker compose:
   Suppose we have 50 microservice. If we want to spawn container for each one of them, then we must individually execute run command for each image and spawn a container for each one of them. This is a time taking process.
   With docker compose with a single command one can spawn containers for all micro services.

   - Structure of docker compose file:

```
version: "3.8"

services:

  accounts:
    image: eazybytes/accounts:latest
    mem_limit: 700m
    ports:
      - "8080:8080"
    networks:
      - eazybank-network

networks:
  eazybank-network:
```

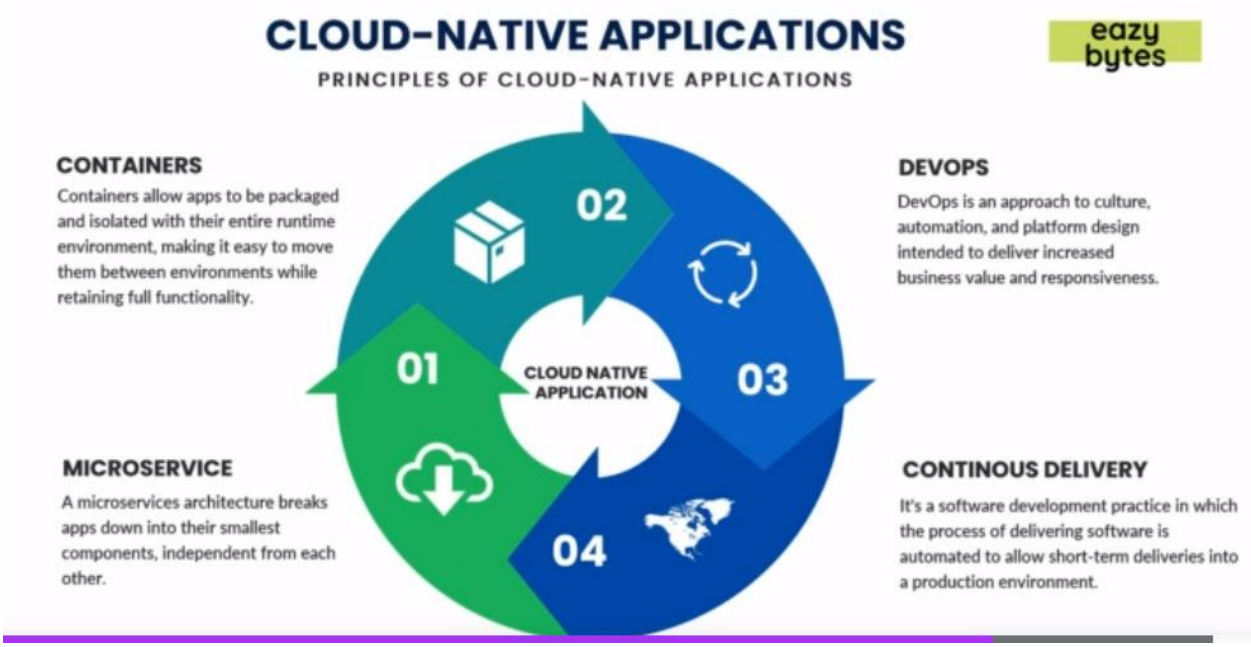
- Services tag gives details of microservices to deploy
- Networks give information of the network shared by Microservices, as you can see all micro services are sharing same network.
- With "-" you can provide multiple entries.
- Image tag is for giving reference of docker image pushed to docker hub

➢ Navigate to folder where docker-compose file is present. Now execute below command to run docker compose file
docker – compose up
To stop containers, execute command:
docker – compose stop

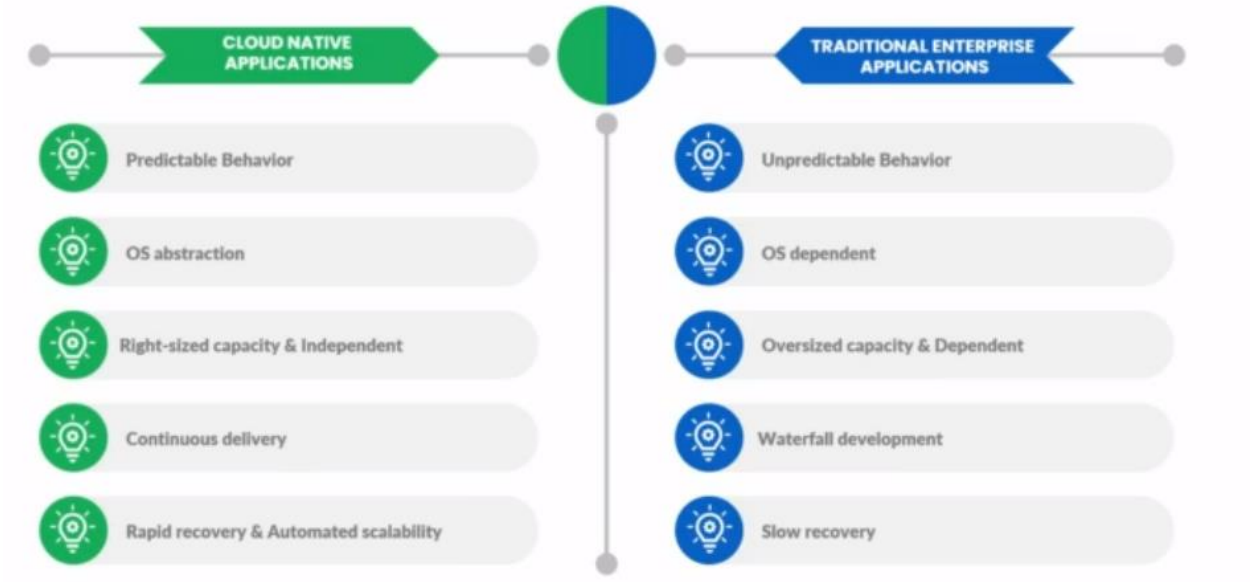
CLOUD NATIVE APPS

1) Definition:

- Applications composed of small modules or services which can serve rapid business changes
- Able to scale up and scale down fast
- If an application is "cloud-native" then its specifically designed to run anywhere: on-prem, AWS, Azure or GCP. So, it's about how applications are developed and deployed not where.
- Main pillar of cloud native apps is Micro services.



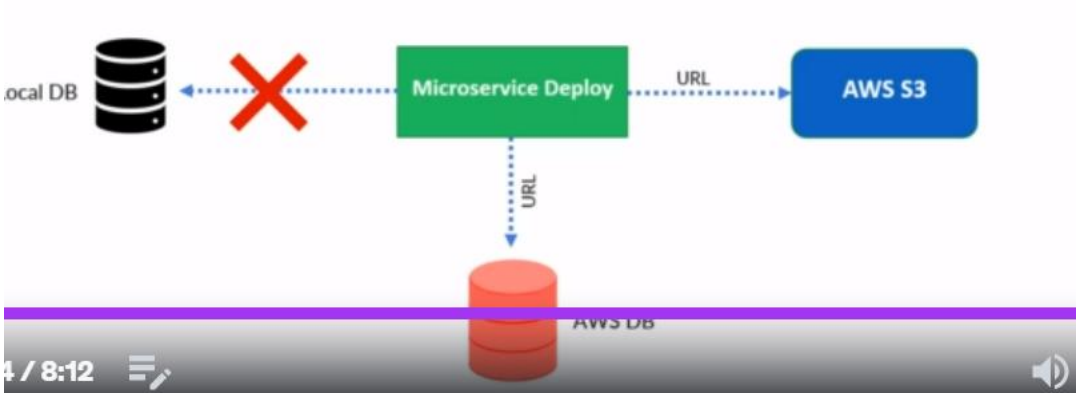
2) Difference between cloud native apps and Traditional apps:



- A docker image can be executed in Windows, Mac, or Linux. Docker will take care of OS abstraction.
- CD means when any change is made then already running services will not be affected, downtime will be negligible.
- Docker containers are right sized, they do not take up entire VM space. In case if more space is needed then that can be achieved through orchestration services like K8s.
- Scalability is very easy. For example, on Sat and Sun we need to spawn extra 10 instances of Accounts MS, then this cannot be easily achieved in traditional application.

3) Twelve Factor App Concepts:
Twelve Factor app lays down principles to be followed to build a cloud native application.

a) Code Base: Each Microservice should have a separate code base in a separate repository. They can be deployed in multiple envs like Dev, Test or Production but each MS will have its own repo. That way each MS can be developed and maintained separately.
b) Dependencies: Jar file dependencies needs to be figured out and explicitly defined using build tools like Maven or Gradle. That way your docker image will contain all the required libraries and you can then use that image anywhere to run.
c) Configuration: You should always store environment specific configuration (for example db details etc.) outside of source code. If we store such configuration inside Micro service code then we might have to change docker image from env to env, this will break philosophy of a cloud-native app.
Hence, we should always keep configuration outside of deployable Microservice, that way if configuration changes then you are not required to re deploy the MS, as service will automatically refer the updated configuration.
d) Backing Services: This principle indicates that microservices should be able to switch connection when deployment env changes with out change in code. For example, if in a given env MS uses local DB then the same MS should be able to run in AWS env where it will use AWS DB without change in code or container or image. This is possible if we keep configuration externalized:



e) Build, Release, run: We should keep build stage separate from Release stage. Foe example we build our code, then based on env to be deployed we choose a specific configuration and create Release.

In essence, again we should build Microservice which should run independent of env in which they are running.

f) Processes: In a typical MS communication, 1 can call 2, 2 will call 3, 3 will then send response to 2 and then 2 will send response to 1. But in this communication never store

data in session of microservice instance (as the same instance may scale down). This is known as Stateless communication. Only share request and response.
If there is still needed to store something, then that should be saved in a database.

g) Port Binding: Each Micro service should have its own port and interface. For example, we generate a spring boot web app using 8080 as port. The same port can be used in all instances of the same microservice, however outside port will be different for each instance.

There can be another MS which is a PHP application running on port 8081. The same port can be used in all instances of the same microservice, however outside port will be different for each instance.

Thus, with Spring Boot Framework and docker commands we can control on which port MS instance is running.

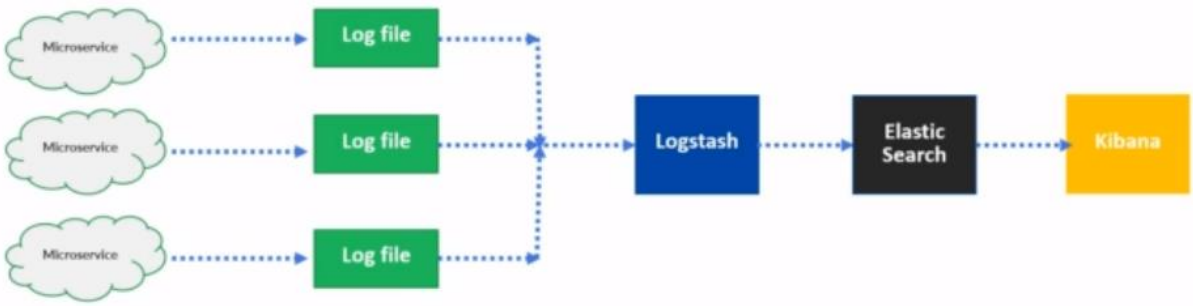
h) Concurrency: This principle states that in situation of heavy load on application we should always go for Horizontal scaling instead of vertical scaling. In other words, no of containers instances should be increased or decreased instead of increasing or decreasing the CPU/RAM etc of the server altogether.

i) Disposability: This principle states that at any point of time microservice instances should be able to gracefully dispose of without affecting the application. For such situations orchestration service like K8s are helpful as they ensure that:

- Start up will be fast in case when scaling is required
- Graceful shutdowns to leave system in consistent state

j) Dev/Prod parity: Dev/Prod/Test env should be similar. This will make sure that entire application is tested correctly. Think of a scenario where Dev configuration is different from Testing, where we did some manual configuration in Dev env. In such cases your application will run in Dev but not in Test.
If you keep same configuration, then you can promote code from Dev to Test fast and this will decrease Testing time as entire application set up has already been tested with similar configuration Dev env.

k) Logs: In a monolithic application its very easy to trouble shoot issues using logs. But in case of MS based architecture there are several of Microservices instances running in multiple servers. In such case there should be a centralized location to stream and analyze logs. For this purpose, ELK stack is used. MS will push logs in the form of event stream in log stash, then it's up to ELK stack to search and analyze logs.



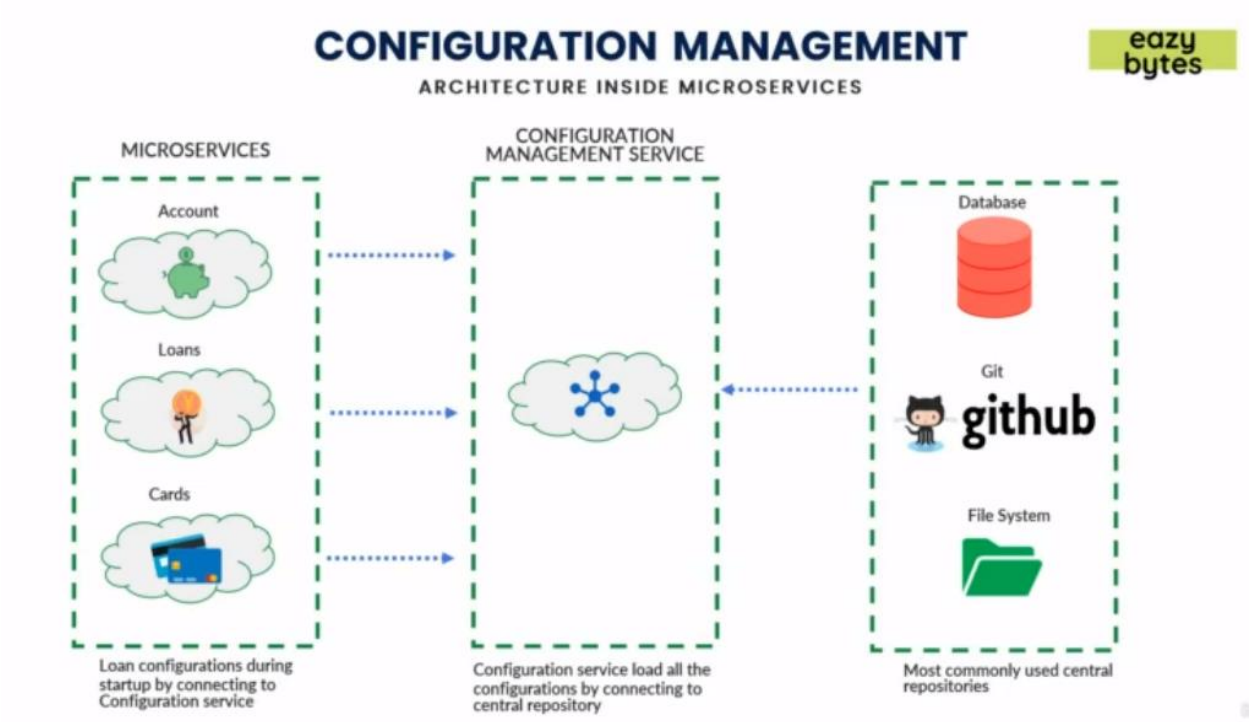
If MS a calls b and b calls c, using ELK stack we can figure out where exactly issue occurred in entire request flow.

l) Admin Process: Admin Process like data clean up, pulling analytics report should be maintained separately from the application. Further these scripts should be maintained in source code repository and should not change form env to env

CONFIGURATION MANAGEMENT IN MICROSERVICES

1) As per the principle, in a cloud native app, configuration details (db details, sftp folder location etc) should be maintained outside of MS code base. Doing so will not require MS code to be changed when configuration information is required to be changed
2) Challenged associated with configuration:
   a) How to externalize configuration
   b) How to inject them in MS
   c) How to maintain configuration information in a way that any change in configuration should not require restart of MS.

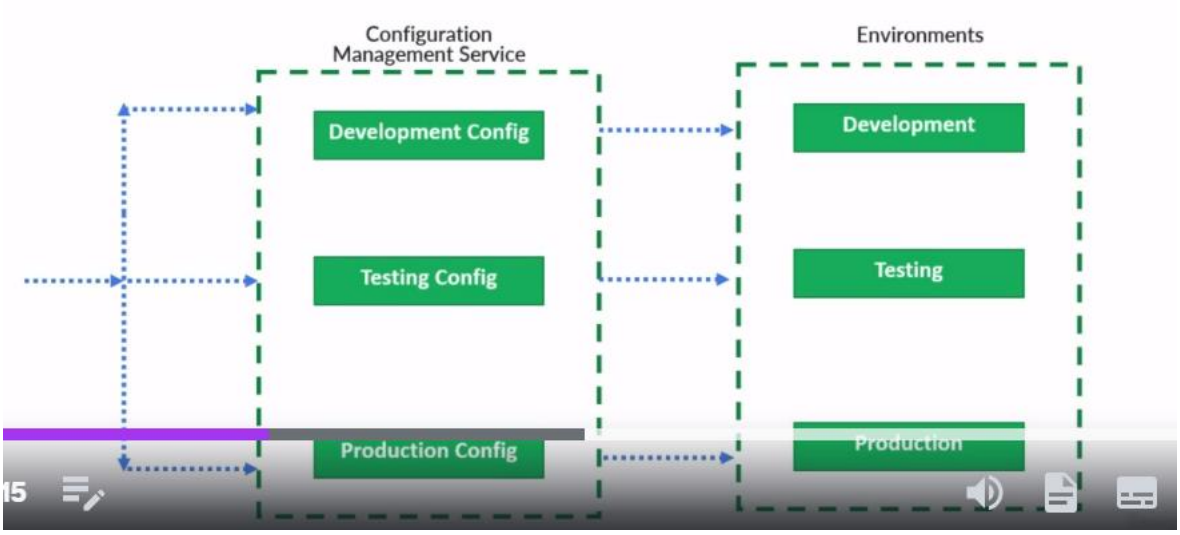
3) Configuration Management Architecture:
   To solve challenges mentioned in step 2), given below architecture can be followed:



- Create a separate MS to hold configuration information
- Configuration Management MS will load configuration information from a central repository
- Central Repository can be Git Hub, File System or DB
- For example, we can have three different configurations for Accounts MS for Dev, Test and UAT env. That way we will have 9 different configurations
- Our main Micro Services will load configuration information through Rest end points exposed by Configuration Management Service
- But lets say we have 50 Microservices, then in that case we need a robust configuration management system in terms of ease, maintenance, and performance.
  To create architecture mentioned above we make use of Spring Cloud Config module of Spring Framework.

4) Spring Cloud Config:

- Spring Cloud Config is a sub project of Spring Cloud.
- Provides server and client-side support for externalized configuration
- Server here refers to server where Configuration Management Service is up and running
- Client here refers to individual micro services
- Create a docker image using your code base
- When you launch the container then based on environment where container is launched, that environment specific configuration information will be load by Microservice using Configuration Microservice. Thus, the running container will behave like a Dev container or Test container or UAT container.



Spring Cloud Config Features:

A) Server-Side Features:

- Expose a HTTP Rest based API via a configuration Micro service. The Microservice can store information in key-value pair or YAML based
- Can also encrypt and decrypt values
- Its easily embedded using @EnableConfigServer annotation

B) Client-Side Features (For your API Micro services):

- You can bind your Micro service with Spring Cloud Server and read environment specific configuration.
- Can also encrypt and decrypt values

5) Building Spring Cloud Config Server/Service

A) Create a new Micro Service Spring Boot service configserver adding Spring Cloud Server and Spring Actuator as dependency

B) In Spring main class add annotation @EnableConfigServer. This annotation will indicate spring framework that this is Config Server application which can read configuration from a centralized repo like Git, Vault or Class path and it can expose configuration through rest end points.

C) But we need to tell Config Server the location from where config information should be loaded. There are three possible locations : File System, Git Hub, Class path.

D) Location of config repo is given in application.properties file

E) Loading configuration information from class path:

- Class path: In this case our Config Server will load configuration information from Class path. For this provide given below entry on Config Service Micro Service's application.properties file:

```
spring.application.name=configserver
spring.profiles.active=native
spring.cloud.config.server.native.search-locations=classpath:/config
```

- Since we have provided class path location as classpath:/config, hence create a folder config under src/main/resources folder. Inside config folder provide configuration for each Microservice for each environment. Since we have three microservice, we need configuration properties for dev, prod and test for each of them. Hence we need total 9 configuration files.

Let's take example of dev env configuration details for accounts microservice. Create a file with name accounts-dev.properties (Similarly make accounts-prod.propeties, file with name accounts.propeties will refer to default environment). A typical configuration information will look like below:

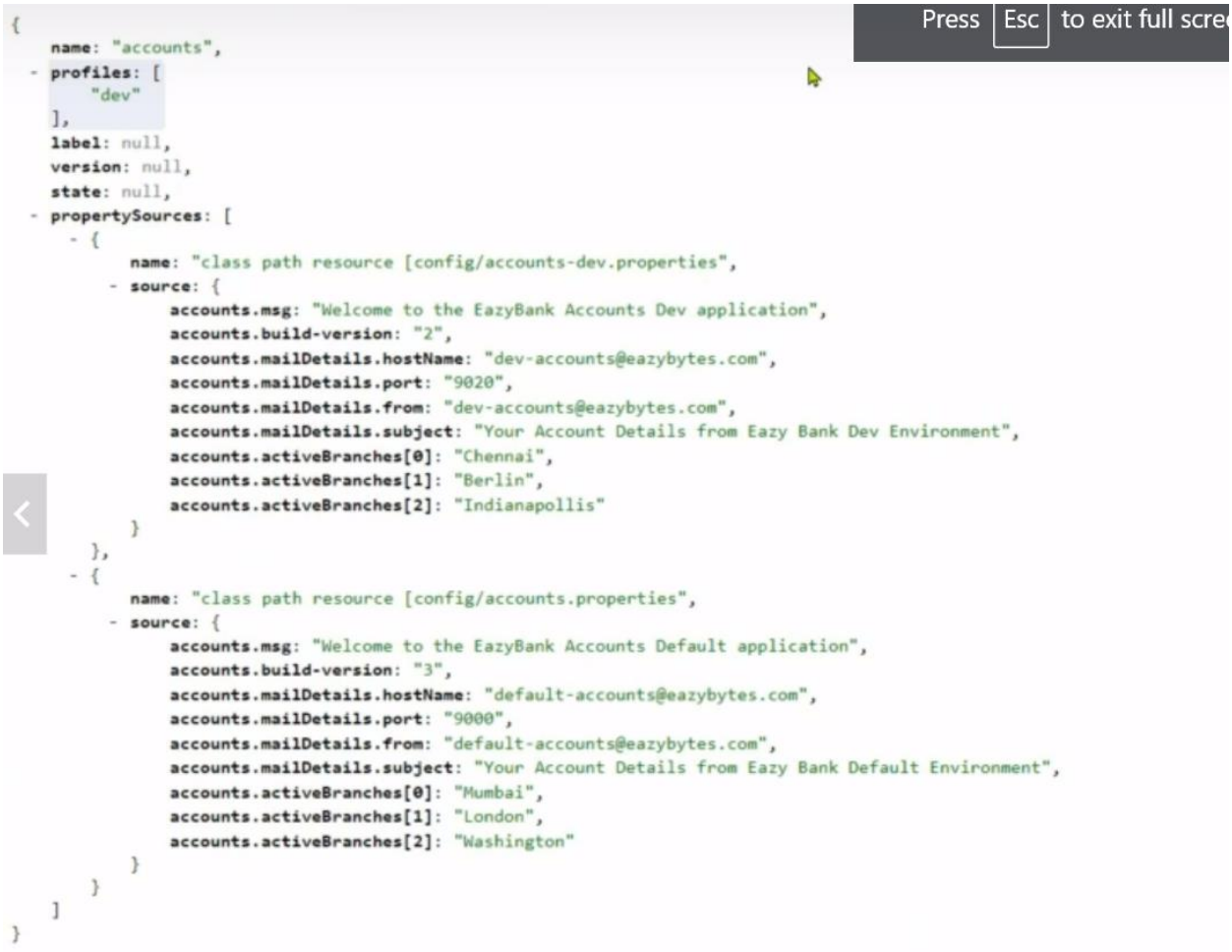
accounts.msg =Welcome to the EazyBank Accounts Dev application
accounts.build-version=2
accounts.mailDetails.hostName=dev-accounts@eazybytes.com
accounts.mailDetails.port=9020
accounts.mailDetails.from=dev-accounts@eazybytes.com
accounts.mailDetails.subject=Your Account Details from Eazy Bank Dev Environment
accounts.activeBranches[0]=Chennai
accounts.activeBranches[1]=Berlin
accounts.activeBranches[2]=Indianapollis

Notice that suffix everywhere is accounts. We can provide properties in the form of arrays and map data structure also.

- Start the configuration management micro service. Once the micro service is up it will expose rest end points which client microservice can use to access configuration information.
- You can check rest end point url using:
  http://<ip>:<port>/<configurationfilename>/<env-name>

  For example, if your microservice is running in local host at port 8071 and you want to access configuration for accounts for dev environment then use url:

  http://localhost:8071/accounts/dev
  you will see response like given below:



Please note that default configuration is also loaded

To access configuration information for accounts microservice for prod environment use url:

http://localhost:8071/accounts/prod

To access configuration information for accounts microservice for default environment use url:

http://localhost:8071/accounts/default

- Now here is the beauty. Change the value of any property. You will see that change will be reflected in rest service response WITHOUT STOPPING THE MICROSERVICE.

F) Loading configuration information from local file system: One may choose this option as we do not want to keep configuration information as part of configuration management service itself, we need to keep it separate outside of microservice. We also do not want to keep it inside Git Hub location to avoid versioning and changes from other developers:

- File location can be from your local file system or from cloud location say AWS S3 bucket.
- Copy config folder from class path to local file system say in C drive
- Keep profile as native only in application.properties file
- Comment the entry:
  spring.cloud.config.server.native.search-locations=classpath:/config

  and provide new entry as:
  spring.cloud.config.server.native.search-locations=file:///C://config

- Now here is the beauty. Change the value of any property. You will see that change will be reflected in rest service response WITHOUT STOPPING THE MICROSERVICE.

G) Loading Configuration from Git Hub location: This is the more preferred and advisable approach to follow:

- Create a repo in Git Hub and load configuration information in that repo
- Change the profile to git in application.propeties file
- Comment the given below entries:
  spring.cloud.config.server.native.search-locations=classpath:/config
  and
  spring.cloud.config.server.native.search-locations=file:///C://config

  Now provide new entries as:

  spring.cloud.config.server.git.uri=<your config repo url>
  spring.cloud.config.server.git.clone-on-start=true
  spring.cloud.config.server.git.default-label=main

6) Building Spring Cloud Configuration Service Clients:
   Configuration Service Clients will be our business Microservices. We need to make change in them so that they will load configuration information upon start up. Lets try to understand how to do that.

A) Change in pom.xml:

- Add a new entry for spring cloud version under property tag:

```
<properties>
        <java.version>17</java.version>
        <spring-cloud.version>2021.0.4</spring-cloud.version>
    </properties>
```

- Add dependency management for spring cloud:

```
<dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

Add above entry after dependencies tag.

- Add dependency for spring cloud config under dependencies tag:

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

- We need to tell microservice the location from where it can read configuration properties. For this add given below three entries in Microservices's application properties file:

```
spring.application.name=accounts
spring.profiles.active=prod
spring.config.import=optional:configserver:http://localhost:8071/
```

- Write code in client Micro service to load configuration information at start up.

```
@Configuration
@ConfigurationProperties(prefix = "accounts")
@Getter @Setter @ToString
public class AccountsServiceConfig {

        private String msg;
        private String buildVersion;
        private Map<String, String> mailDetails;
        private List<String> activeBranches;

}
```

This is how you have a version of Microservice which will read configuration information on service start up from an external resource. The beauty is that when you change configuration information then you are not at all required to change code and reboot either client or config MS. Hence same docker file can still be used upon configuration information change.

Similarly, you can microservice/docker image version for dev and uat env/profile as well. But this means there will be a different version of docker file and image if we move from one env to other env. But there is solution to this as well.

7) Docker Compose File change:
We have seen that we can externalize configuration information using a separate configuration service. Business Micro services will read configuration information configuration service at application start up. But in this approach, profile/env information is inside Microservice. So, if we want to change the environment then, we have change code and regenerate docker image.

Using docker compose file, we can provide profile information in docker compose file thereby same docker image can be used for different environments. Only docker compose file must change.
Follow given below steps to implement this:

A) Regenerate jar files for all business micro services and push docker image to docker hub. Remember do not change any entry in application.properties file. Although we will externalize profile related information in docker compose file, even then do not remove given below entry from application.properties file

```
spring.application.name=accounts
spring.profiles.active=prod
spring.config.import=optional:configserver:http://localhost:8071/
```

Reason being our client micro services are still annotated to read configuration file so at application build time these properties will be sought after.

B) Create docker compose file with given below information:

```
accounts:
  image: eazybytes/accounts:latest
  mem_limit: 700m
  ports:
   - "8080:8080"
  networks:
   - eazybank
  depends_on:
   - configserver
```

```
deploy:
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s
environment:
  SPRING_PROFILES_ACTIVE: default
  SPRING_CONFIG_IMPORT: configserver:http://configserver:8071/
```

- Please note that all micro service will share the same network for creating a bridge.
- Provide entries for all other micro services in docker compose file similarly.
- At application start time, using docker compose information AccountsServiceConfig will load configuration information from config server using information from docker compose file.

8) Refreshing properties: If we change property specific to any env/profile, we need to make sure that all micro service instances should refer latest value without restarting them. For this spring cloud confiig provide a special annotation @RefreshScope. This annotation will expose a new endpoint in spring actuator. When we invoke that url for individual micro services then all micro service will reload their configuration information without restarting them.

Apart from this also add given below entry in micro service's application.properties file:

management.endpoints.web.exposure.include=*

This will expose all end points urls for spring actuator.

A) Start the config server
B) Start any one micro service
C) Get accounts properties
D) Now make change in config properties
E) Get account properties, you will see that you will still get old value
F) Hit below mentioned refresh url:
   http://<ip>:<port>/actuator

   Here you will see refresh url for your micro service.

   Hit that url using post request (without providing any body)

   This will make your micro service to re-fetch config information from config server.
G) Get accounts properties, you will not get updated value.

H) Please note that properties like db details, email will still require server restart as such properties require a re-connection to db or SMTP server.
I) One can create a shell script file or decompose file to hit restart urls for all microservices on config information update instead of manually calling refresh urls for all micro services

9) Encryption and decryption of properties.
Some time we may have a requirement to encrypt certain properties. Follow below steps:

A) Provide given below entry on config server application.properties:

encrypt.key=hitesh791

You can provide any key.

B) Once you this then config server will expose two given below POST urls:

http://<ip>:<port>/encrypt.

Use this url to encrypt value of any property.

http://<ip>:<port>/decrypt

C) After encryption replace original value with encrypted value.
But provide a suffix {cipher}
This suffix will tell spring clound config to decrypt the value before sending it
D) Hit url to get property value from business micro service. You will see that spring cloud config automatically takes care of decrypting value and sending it to your business micro service.

SERVICE DISCOVERY AND REGISTRATION

1) Introduction:

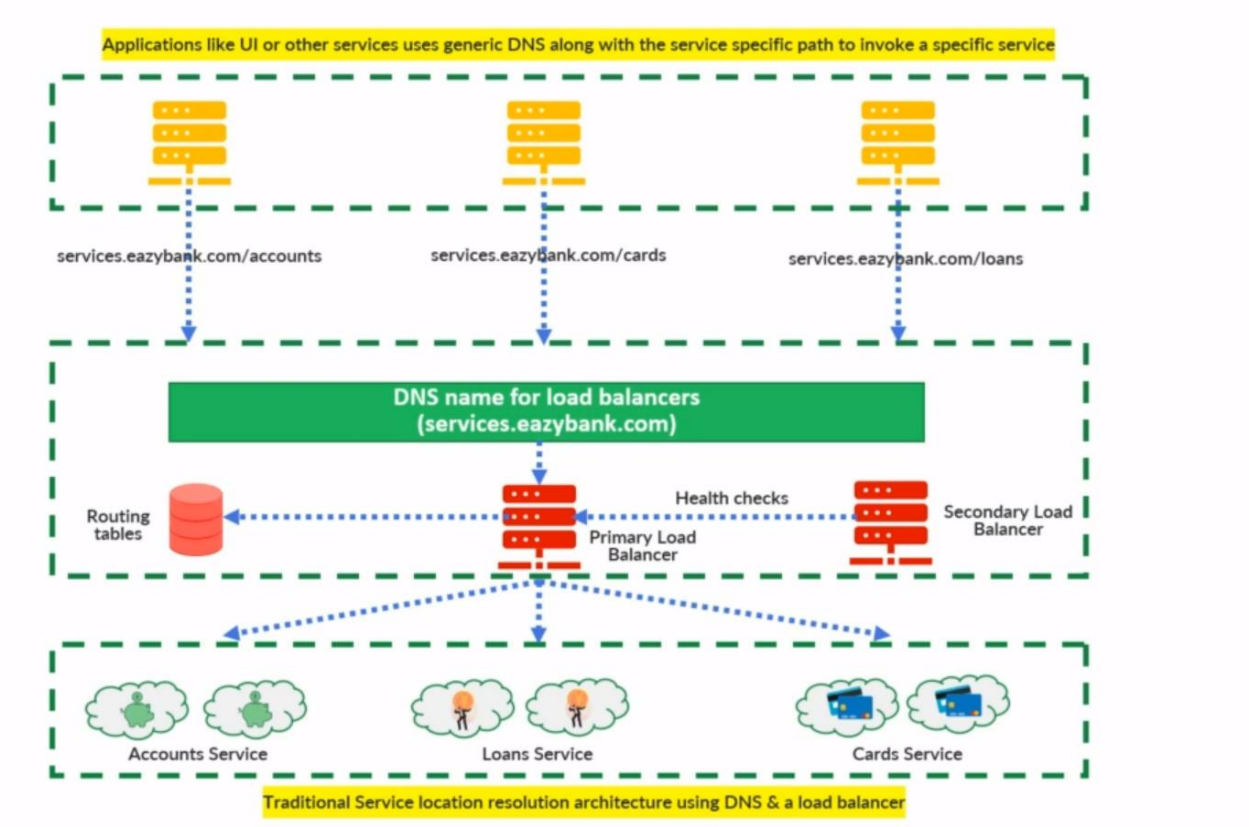
A) We can have multiple microservices for an application
B) These micro services share a network
C) IP and Address and port of these micro services are dynamic in nature, as instances can be scaled up or down
D) In such case if microservice want to communicate with one another then how they will know end point address
E) What about load balancers for instances of a given micro service.
F) How a new instance of a given micro service will enter the same network.

G) How other micro services will know that a new instance is spawned and its ready to service the request.

All these challenges are solved by Service Discovery and Registration pattern:

A) Service Discovery and Registration pattern solves all above problems
B) It contains a central server which maintain a global view of all addresses
C) Micro services or clients when they start then they register (ip, port, end point and details) themselves in central server
D) MS/clients also send heart beat at regular interval to central server to inform about their health. If any of instances is slow and not able to send heart beat signal in a given time then all of its details are removed from central server.
E) Thus this pattern helps in maintaining a healthy network topology in a micro services based architecture.

2) Why we cannot use Traditional Load Balancers:
A traditional load balancer looks like below:

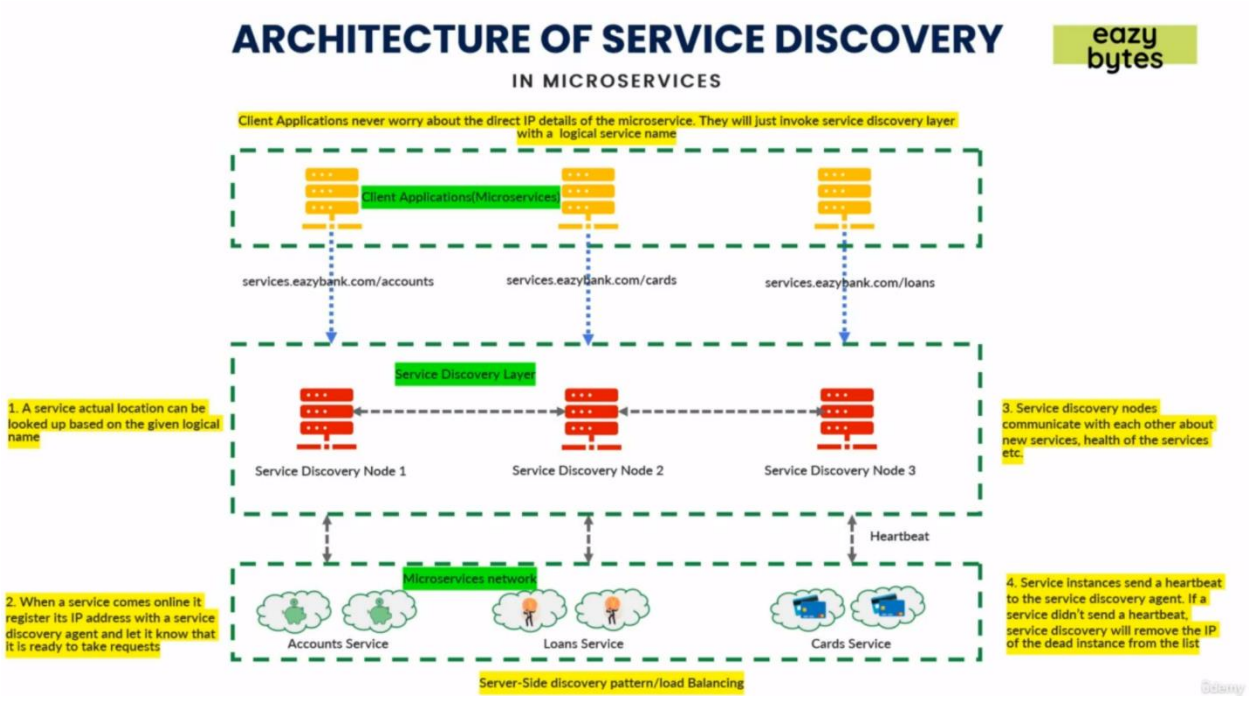


A) You invoke a specific service using a generic DNS name to invoke any service like services.eazybank.com/accounts
B) The DNS maps to a load balancer end point
C) Load Balancer queries routing table to get available IP addresses or location for the invoked service.
D) Load Balancer then selects any one ip based on geographic location or round robin etc.
E) Apart from this there is Secondary Load Balancer
F) Primary Load Balancer sends Heat beat signal to Secondary.
G) If Secondary finds that Primary is not responding, then secondary comes into action immediately.
H) All clients invoke request to Load Balancer only (whose IP address does not change). While internal service IP address keep on changing.

This approach cannot be used in Micro service-based architecture:

A) In a micro service based architecture, ip addresses of services keeps on changing very dynamically
B) This approach is not container friendly, imagine maintaining instances of thousands of micro services.
C) This cannot be scaled horizontally, what if we want to manage a cluster of Load Balancers.

3) Architecture of Service Discovery Inside Micro Services:



A) Service Discovery Layer:

- As a developer you will first build your Service Discovery Layer.
- Service Discovery Layer is a micro service
- In Service Discovery Layer we will have three nodes or agents.
- Just like we had a configuration micro service to hold configuration information, this micro service will hold individual micro services information (ip, end point, etc.)
- Service Discovery Nodes also communicate with each other to share information like micro services instances, health etc. This makes sure that all service Discovery Nodes have same information. This peer to peer communication is known as gossip or infection protocol
- Once they are built, then start this micro service.

B) Micro service network:

- Once Service Discovery micro services is up then start instances of individual micro services
- Upon start up these micro services instances will register their address details with Service Discovery agent.
- They will register themselves with a logical name, for example account micro service will register with /accounts, cards micro service will register with /cards etc.
- Hence Service Discovery micro service will have information of what are all the micro services available in the network and what are their instances.
- Each individual micro service instance also send heart beat signal to service discovery node (any node). They have to send heart beat within 30s by default. If they are not able to do so then, service discovery nodes will wait for 90s (by default, 3 chances) to send heart beat signal. After that Service Discovery node will remove information of that instance.

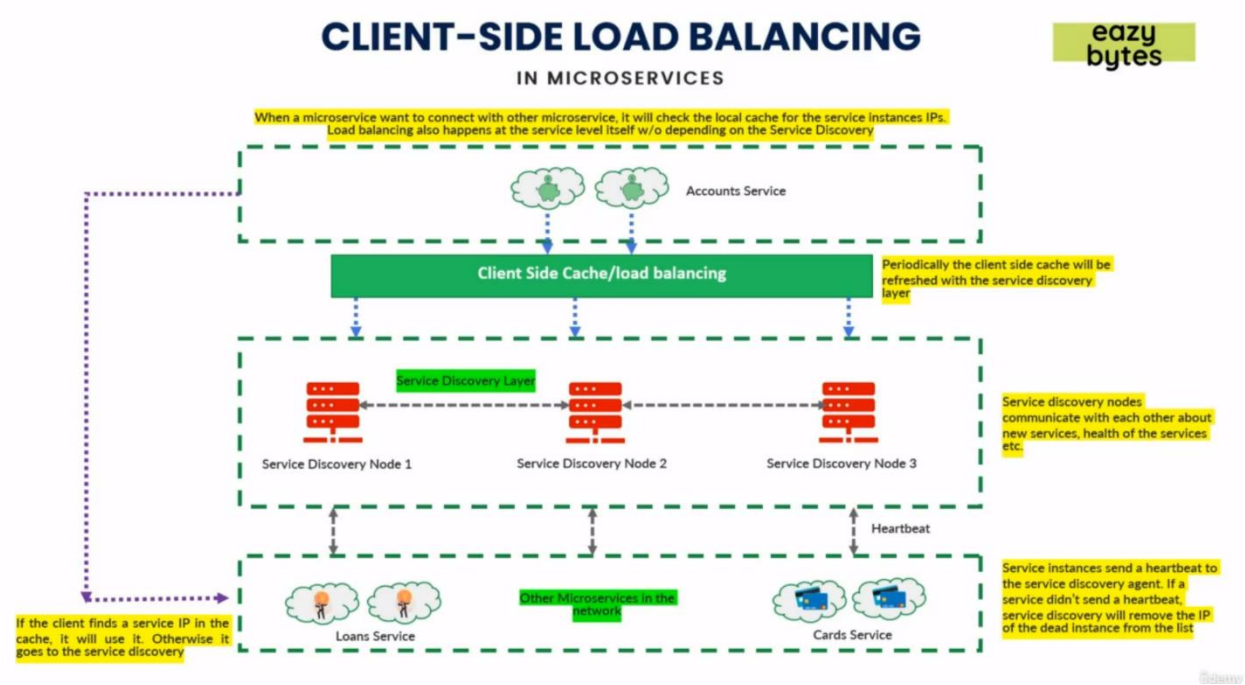
C) Client Micro services: Here, we are talking about back end micro services invoking other backend micro service i.e. internal micro service communication.
(For Micro service invocation from UI we have a different concept of API gateway)

- Client micro service invoke any other micro service by calling service discovery layer. For example a client micro service will invoke accounts micro service by invoking service discovery layer using url services.eazybank.com/accounts.
  Service Discovery Layer will check available instances for accounts micro services. It will find that if multiple instances are available then it will route request to a specific instance using some algorithm (for example round robin)

This over all approach is known as service side load balancing:

- Service Discovery consists of a Service Registry (a key- value store) and an API to write and read from.
- Client interact with service registry to know exact location of mico service instance on network
- This approach had HA as we can have multiple service discovery nodes
- There is peer to peer communication between service discovery nodes
- Ips can be dynamically managed.

4) Client-Side Load Balancing:



- We use Client-Side Load Balancing to cache exact end point url information for a service
- This also reduces load on Service Discovery Layer
- Lets try to understand this with a flow:
    - Accounts Service requests Loans Service for the first time
    - There is no information for Loans Service in Client-side cache
    - It invokes Service Discovery Layer to get available instances and returns an instance
    - Client side cache synchronizes Loans service instances (all available) info from Service Discovery Layer
    - Accounts service request requests Loans service once again
    - This time information for Loans Service will be found in Client Cache
    - Client side Load Balancer will then return a specific instance based on some logic (round-robin or proximity)
- Client side cache will always refresh information from Service Discovery layer in a configurable periodic manner
- But there may arise a scenario when Client side cache can return an instance end point address which is down. In that case instead of failing Client Micro service will query Service Discovery layer to get latest instances information. Client side cache will also refresh its information
- All these is achieved by Spring Boot Framework with minimal configuration and coding.

5) Spring Cloud Support for Service Discovery and Registration:
   Spring Cloud provides this support using three components:

   A) Spring Cloud Netflix Eureka Server (Eureka Server): Service Discovery Agent which act as Service Registry
   B) Spring Cloud Load Balancer Library for client side-Load Balancing
   C) Netflix Feign client: For performing operation of Service Discovery.

6) Spring Boot Support for Eureka Server:
   A) Provide @EnableEurekaServer annotation in main class. This will tell Spring Boot Framework that this Micro service will act as a Eureka Server.
   B) Add given below properties in application.properties file:
      spring.application.name=eurekaserver
      spring.config.import=optional:configserver:http://localhost:8071/
   C) Add Eureka Server properties in config repository:
      server.port=8070

      eureka.instance.hostname=localhost
      eureka.client.registerWithEureka=false
      eureka.client.fetchRegistry=false
      eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/

Please note that above information can also be provided in application.properties file of Eureka Server Micro service.

Once you start the Eureka Server application, it will check if spring.config.import property is available, if Yes then it will connect to Config Server and read relevant properties and start the Eureka Server (Otherwise it will read information available in application.propeties file to get details of port, end point address etc.)

Once its up, hit the url http:< eureka.instance.hostname>:server.port

In our example cas Eureka server will be available at http://localhost:8070

7) Make changes in Client Micro service to register with Eureka Server:
   Now we need to make change in our client Micro service so that anytime when instance is launched it will register itself with Eureka Server. Follow given below steps:

   a) Add given below entries in pom.xml:
      <dependency>
                  <groupId>org.springframework.cloud</groupId>
                  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
            </dependency>

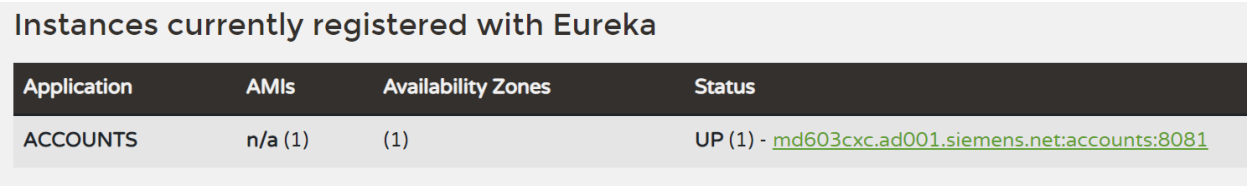
b) Add given below entries in application.properties file of client application:

```
eureka.instance.preferIpAddress = true
eureka.client.registerWithEureka = true
eureka.client.fetchRegistry = true
eureka.client.serviceUrl.defaultZone = http://localhost:8070/eureka/

## Configuring info endpoint
info.app.name=Accounts Microservice
info.app.description=Eazy Bank Accounts Application
info.app.version=1.0.0
management.info.env.enabled = true

endpoints.shutdown.enabled=true
management.endpoint.shutdown.enabled=true
```

c) Start the config server, Eureka Server and client micro service application. Now hit the url for Eureka Server, you can now see the information regarding running micro service instance.



Hit the highlighted url. Upon hitting it wil take you to actuator url where information will be displayed as per info.app..* properties values in application.properties file

Important Points:

- Some times it takes time to get information reflected Eureka Server. So wait for some time before information of your instance gets displayed in Eureka Server
- Right now we have only one Accounts (name comes from spring.application.name property value) Micro service instance. Hence no of AMIS is 1. If we start multiple instances then there will be multiple entries to Micro service with logical name Accounts where each entry will correspond to each instance.
  So as user has to invoke Eureka Server url with logical name of the micro service to get details of available instances of that micro service.
  For example,hit the url:
  http://<euraka server ip>:<eureka server port>/eureka/apps/<service name>

For example in our case, hit the url:
http://localhost:8070/eureka/apps/accounts
Then given below xml will be displayed:

<application>
<name>ACCOUNTS</name>
<instance>
<instanceId>md603cxc.ad001.siemens.net:accounts:8081</instanceId>
<hostName>192.168.29.174</hostName>
<app>ACCOUNTS</app>
<ipAddr>192.168.29.174</ipAddr>
<status>UP</status>
<overriddenstatus>UNKNOWN</overriddenstatus>
<port enabled="true">8081</port>
<securePort enabled="false">443</securePort>
<countryId>1</countryId>
<dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
<name>MyOwn</name>
</dataCenterInfo>
<leaseInfo>
<renewalIntervalInSecs>30</renewalIntervalInSecs>
<durationInSecs>90</durationInSecs>
<registrationTimestamp>1671385333253</registrationTimestamp>
<lastRenewalTimestamp>1671385333253</lastRenewalTimestamp>
<evictionTimestamp>0</evictionTimestamp>
<serviceUpTimestamp>1671385333253</serviceUpTimestamp>
</leaseInfo>
<metadata>
<management.port>8081</management.port>
</metadata>
<homePageUrl>http://192.168.29.174:8081/</homePageUrl>
<statusPageUrl>http://md603cxc.ad001.siemens.net:8081/actuator/info</statusPageUrl>
<healthCheckUrl>http://md603cxc.ad001.siemens.net:8081/actuator/health</healthCheckUrl>
<vipAddress>accounts</vipAddress>
<secureVipAddress>accounts</secureVipAddress>
<isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
<lastUpdatedTimestamp>1671385333253</lastUpdatedTimestamp>
<lastDirtyTimestamp>1671385333203</lastDirtyTimestamp>
<actionType>ADDED</actionType>
</instance>
</application>

You get other useful information regarding instance like, ip, port, health, actual url etc.

- Eureka server exposes Rest urls to access information of registered micro services.

- By default all client micro services will send hear beat signal to Eureka Server every 30s

8) De- registering micro service instance:

   a) The actuator info for each micro service will expose a shut down url (because we enabled this url in application.propeties file)
   b) Hit the url: http://<ip address of ms>:<port of ms>/actuator. In our case hit url for accounts micro service:
      http:localhost:8081/actuator

      Here you will also see a shutdown url:
      http://localhost:8081/actuator/shutdown. Hit that url through Post request in Post man. You will observe that accounts micro service instance will de-register itself.

   c) If instance is shutdown forcefully then, Eureka Server wil use concept of Heat Beat to deregister any instance.

9) Feign Clients to invoke other micro services: Now all client micro services are able to register themselves in Eureka Server, its time to learn how individual micro services can discover instances of other micro services using Feign Clients:

Let take scenario where we will expose a new Rest API in Accounts micro service which will fetch account, loan and card details of a customer. Hence in this case Accounts micro service has to invoke loan and card APIs to get required information. Follow given below steps:

a) Add given below entry in pom.xml:

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

   Add this dependency only in ms which calls others ms.

b) Add given below annotation in application main class:
   @EnableFeignClients

c) Just like in JPA Repository we only added interface and rest of the code was automatically generated by JPA Framework. Similarly here also we will only create a client code and rest of the code will be generated by Feign client FW. For example, accounts ms has to invoke client ms code so we will create a Feign client for client ms in accounts ms. See below:

```
@FeignClient("cards")
public interface CardsFeignClient {
   @RequestMapping(method= RequestMethod.POST, value="myCards",consumes = "application/json")
   List<Cards> getCardDetails(@RequestBody Customer customer);
}
```

Similarly develop Feign client for fetching Loans details

d) Develop an API in Accounts micro service to fetch details of accounts, loan and cards by customer id

10) Make changes in docker compose file for eureka server: (Here we have taken example of default docker compose file)
    a) Make entry for eureka server registry:

```
eurekaserver:
  image: hitesh791/eurekaserver:latest
  ports:
    - "8070:8070"
  networks:
    - hitesh791-network
  depends_on:
    - configserver
  deploy:
    restart_policy:
      condition: on-failure
      delay: 15s
      max_attempts: 3
      window: 120s
    resources:
      limits:
        memory: 700m
  environment:
    SPRING_PROFILES_ACTIVE: default
    SPRING_CONFIG_IMPORT: configserver:http://configserver:8071
```

    b) Make changes for client service definition in docker compose file. (Here we have taken example of accounts micro service). Add given below two entries:

```
depends_on:
  - configserver
  - eurekaserver
```

……

```
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE: http://eurekaserver:8070/eureka/
```

11) Running Account Micro service with two instances using docker compose file:

a) Add one more entry for account microservice in docker compose file:

```
Accounts1:
  image: hitesh791/accounts:latest
  ports:
    - "8080:8080"
  networks:
    - hitesh791-network
  depends_on:
    - configserver
    - eurekaserver
  deploy:
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
      window: 120s
    resources:
      limits:
       memory: 700m
  environment:
    SPRING_PROFILES_ACTIVE: default
    SPRING_CONFIG_IMPORT: configserver:http://configserver:8071/
    EUREKA_CLIENT_SERVICEURL_DEFAULTZONE: http://eurekaserver:8070/eureka/
```

- Note that to run another instance of same micro service we have changed service name in docker compose file as in a docker compose file we cannot have two services with same name. But both of them will refer same docker image, hence they both belong to same Accounts Micro service (as per value of property spring.application.name). Hence once there are two instances and they are up they will be registered under same micro service in Eureka Server
- Once both of these instances are up then you will see two instances under Account Micro server, see screen shot below:

Application	AMIs	Availability Zones	Status
ACCOUNTS	n/a (2)	(2)	UP (2) - 9a46a60192ba:accounts:8080 , 47284b08f740:accounts:8080

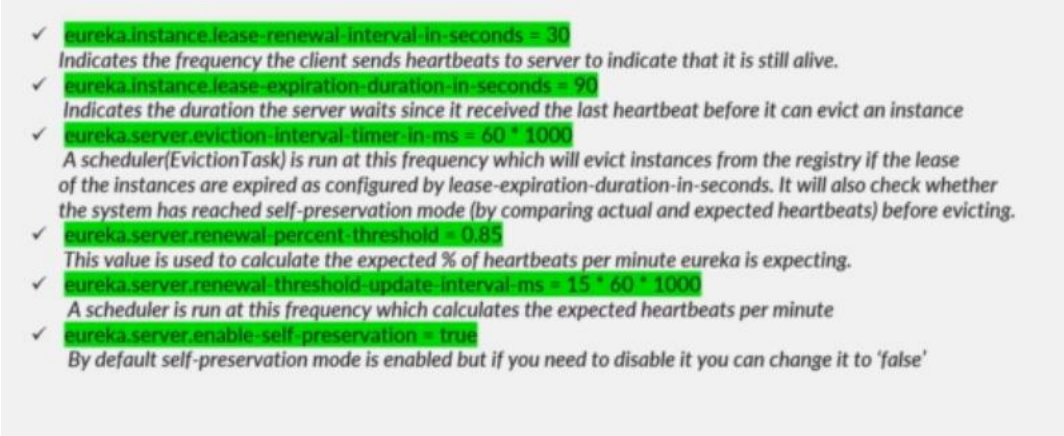
11) Eureka Self Preservation Mode:

A) Eureka Self Preservation mode is a feature of Eureka to deal with network glitches.

B) Under self-preservation mode if no of available instances falls below threshold then Eureka will not deregister any more instances if they stop sending hearth beat signal.

C) For example lets say we have 5 instances and two instances stop sending heart beat signal then Eureka server will have deregister them. But it might happen that those two instances were alive and healthy but they failed to send hearth beat signal due to network issue.
Now we have 3 instances out of 5 which is less than 85% of threshold. Hence if now the remaining 3 instances fail to send heart beat signal then Eureka Server will not deregister them. Eureka server is intelligent enough to deduce that there might be some network glitch.

D) Without this feature we will end up having 0 instances in case of network error even when they were alive and healthy
E) This will also ensure that communication within client micro service can work as they can communicate via their local cache registration details
F) Self preservation mode never expires until and unless down micro services are brough up or network issue is resolved
G) Configurations which can affect Self Preservation:



First two properties are configured at client micro services, remaining are configured at Eureka server.

MAKING MICRO SERVICES RESILIENT

Resiliency is a feature which enables a system to come out of an erroneous or failure situation. In a typical architecture there can be 100s of microservices running, how we can make sure that we will get the desired collective output in case of any failure situation.

Given below such situations can occur:

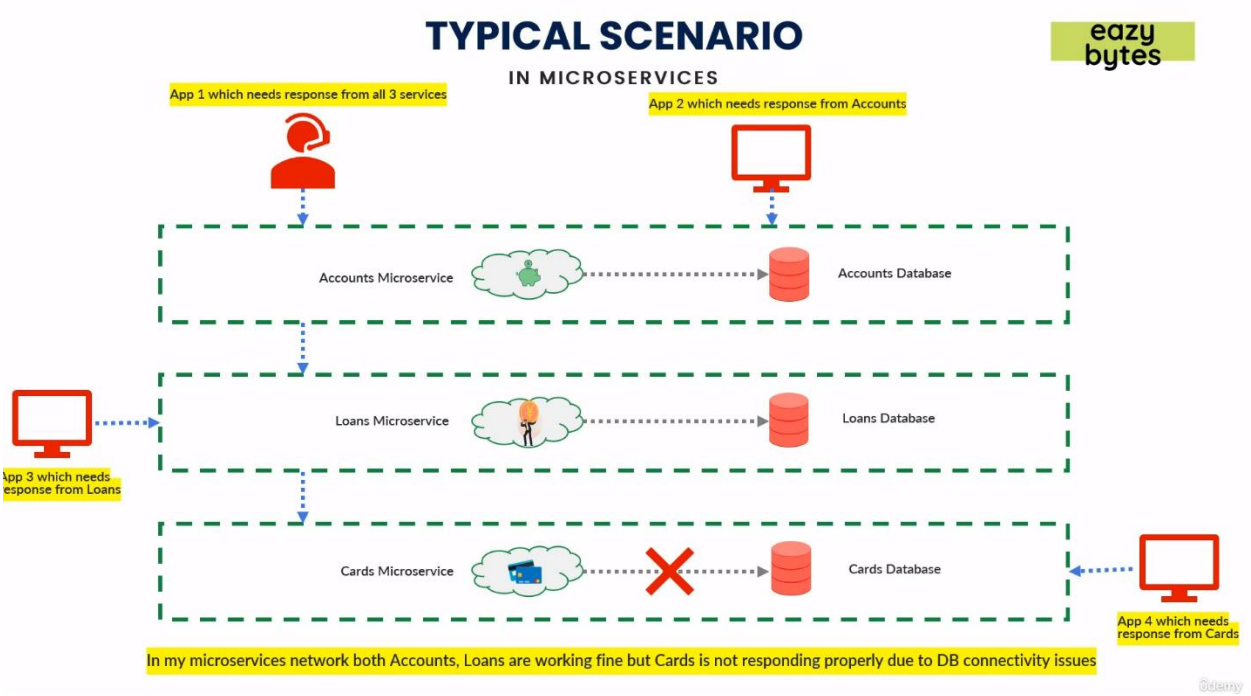
A) Cascading Failure: One slow or failed microservice should not fail the entire chain of microservices
B) Handling Failed micro service gracefully: Building a fallback mechanism when one of the microservices fail. For example, returning a default value of fetching data from DB or calling another micro service or getting value from cache
C) Making Services Self Healing: In the case of slow performing microservices, how we should configure time outs, retries, and give slow micro service some time to recover (during this time slow micro service should not accept any other request)

Spring provide support for above issues through Resilience4j which is a light weight framework inspired from Netflix Hystrix. Resilience4j provide fault tolerance mechanism in event of network or a micro service failure through 4 patterns:

- Circuit Breaker
- Fallback

- Retry
- Rate Limit
- Bulkhead

1) Typical Use case/ Scenario for Resiliency:



A) App1 needs customer details, hence it invokes Account Micro Service API which in turn invokes Loans and Cards Microservice
B) App2 needs response from Accounts
C) App3 needs response from Loans
D) App4 needs response from Cards
E) Cards Microservice started responding slowly due to DB connectivity issue with Cards DB
F) Due to this response from App1 becomes slow
G) Now multiple customers invoke App1. Due to this, multiple threads are created as previous thread are still waiting for response from Cards
H) This will result in multiple threads creation and resource utilization will increase
I) Due to increase in usage of memory, resources and CPU, accounts and loans micro service will become slow.
J) The entire system will hang now.

Above is typical example of Cascading Failure

2) Circuit Breaker Pattern:
   Take example of previous use case where Cards MS is responding slowly due to DB connectivity issue. In such cases Circuit Breaker pattern can be used to avoid Cascading failure.

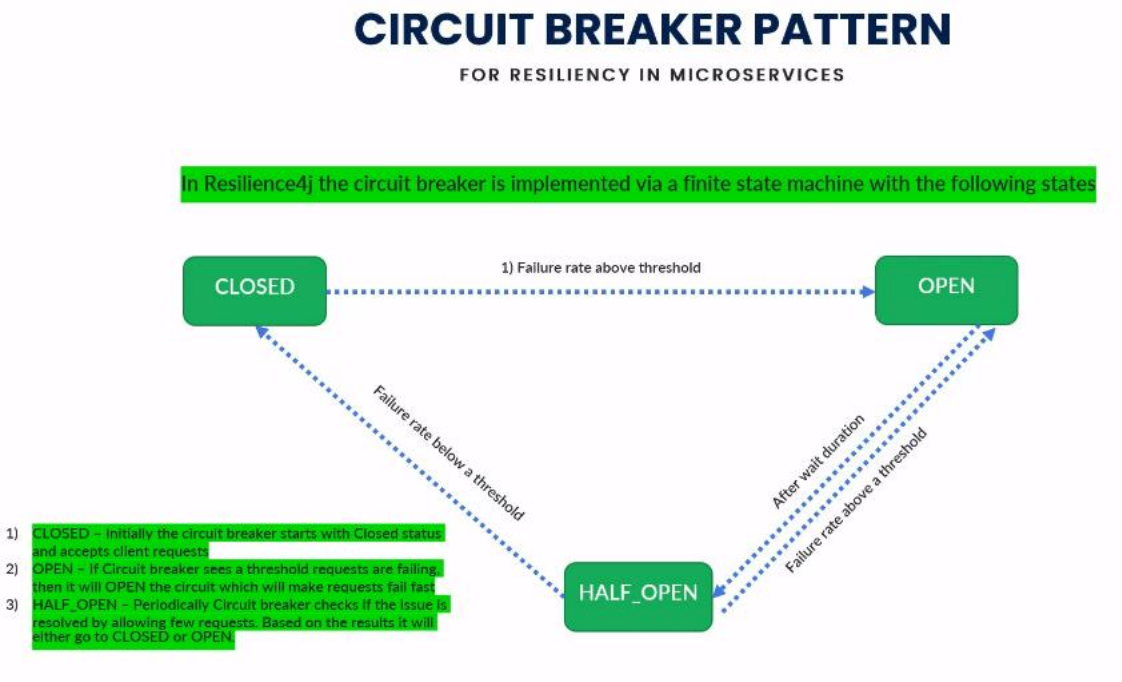
   How it works:

   A) It is inspired from electric circuit breaker. In a electric circuit breaker when too much current passes then circuit is opened so that no more current can pass.
   B) Similarly in software circuit breaker pattern if a call takes too long then that call is killed. Also, if there are too many calls to the service then it can also prevent future calls being made to a service. Thereby we can manage load in system and we can avoid cascading failures.
      When CB is open (it means network is open) then Account and Loans Micro service will respond immediately instead of waiting, also they can receive default response if that is configured. If default response is not configured then they will receive exceptional response in a graceful manner

   C) CB pattern is smart enough to monitor the state of erroneous service. In our case cards ms network is opened as it was responding slowly. By the time when no new requests are sent to cards ms, it can leverage this time to heal itself.
      CB will then periodically check the state of cards microservice. It will half open the network, if the problem is resolved then it will close the network otherwise it will again open the network.
      This is a continuous process which will happen in CB pattern

   D) Advantages of CB pattern:

- Fail Fast: Fails fat instead of waiting
- Fail gracefully: Default response can also be configured
- Recover seamlessly: Gives time to erroneous service to recover.



E) Implementing CB Pattern:

- Add pom.xml dependency in Micro service where you want to apply CB pattern.

```
<dependency>
		<groupId>io.github.resilience4j</groupId>
		<artifactId>resilience4j-spring-boot2</artifactId>
		</dependency>
		<dependency>
		<groupId>io.github.resilience4j</groupId>
		<artifactId>resilience4j-circuitbreaker</artifactId>
		</dependency>
		<dependency>
		<groupId>io.github.resilience4j</groupId>
		<artifactId>resilience4j-timelimiter</artifactId>
		</dependency>
		<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-aop</artifactId>
		</dependency>
```

We want to make Accounts Micro service API myCustomerDetails (which get details from Accounts, cards and loans) resilient, hence above entry in pom.xml of Accounts MS

- Add annotation for CB pattern. In our case add given below annotation to myCustomerDetails API

```
@PostMapping("/myCustomerDetails")
@CircuitBreaker(name="detailsForCustomerSupportApp")
public CustomerDetails myCustomerDetails(@RequestBody Customer customer) {…}
```

We have also given name for Circuit Breaker.

Please note that we can add this annotation at individual API (method) level or Micro service level. But never forget to provide a name for the CB so that you can know which CB has the issue

- Add CB related configuration properties in application.properties file:

```
resilience4j.circuitbreaker.configs.default.registerHealthIndicator= true
resilience4j.circuitbreaker.instances.detailsForCustomerSupportApp.minimumNumberOfCalls= 5
resilience4j.circuitbreaker.instances.detailsForCustomerSupportApp.failureRateThreshold= 50
resilience4j.circuitbreaker.instances.detailsForCustomerSupportApp.waitDurationInOpenState=
30000
resilience4j.circuitbreaker.instances.detailsForCustomerSupportApp.permittedNumberOfCallsIn
HalfOpenState=2
```

F) Testing CB Pattern:

- Start all Micro service except Cards
- Hit Eureka Server url to check if required services are up:

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS	n/a (1)	(1)	UP (1) - host.docker.internal:accounts:8081
LOANS	n/a (1)	(1)	UP (1) - host.docker.internal:loans:8090

We can see that cards MS is not up

- Now call myCustomerDetails API from postman. (Remember this API will fetch data from Accounts, Loans and Cards MS, but cards MS is down)
- You will observe that upon calling, you will receive internal server error as Cards MS is down.


```
{
    "timestamp": "2023-01-17T15:23:55.504+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/myCustomerDetails"
}
```

- In console you will receive error like

  feign.FeignException$ServiceUnavailable: [503] during [POST] to [http://cards/myCards] [CardsFeignClient#getCardDetails(Customer)]: [Load balancer does not contain an instance for the service cards]
  at feign.FeignException.serverErrorStatus(FeignException.java:256) ~[feign-core-11.10.jar:na]

- Invoke the same call from postman 4 more times. After total of 5 calls you will see that response from server will be fast and exception in console will be different:

  io.github.resilience4j.circuitbreaker.CallNotPermittedException: CircuitBreaker 'detailsForCustomerSupportApp' is OPEN and does not permit further calls

  We can now see that after 5 calls, CB has opened the circuit for Cards Microservice and error now changed in console.

- Hit the actuator url: http://localhost:8081/actuator/circuitbreakerevents
  You will observe that this will show that initial calls took time and after that response was fast with a different exception message:

```
object{1}
   circuitBreakerEvents[9]
      0{6}
         circuitBreakerName:detailsForCustomerSupportApp
         type:ERROR
         creationTime:2023-01-17T20:53:55.377756400+05:30[Asia/Calcutta]
         errorMessage:feign.FeignException$ServiceUnavailable: [503] during [POST] to [http:/
         durationInMs:2711
         stateTransition:null
      1{6}
         circuitBreakerName:detailsForCustomerSupportApp
         type:ERROR
         creationTime:2023-01-17T20:55:41.162334400+05:30[Asia/Calcutta]
         errorMessage:feign.FeignException$ServiceUnavailable: [503] during [POST] to [http:/
         durationInMs:60
         stateTransition:null
      2{6}
         circuitBreakerName:detailsForCustomerSupportApp
         type:ERROR
         creationTime:2023-01-17T20:55:43.801531700+05:30[Asia/Calcutta]
         errorMessage:feign.FeignException$ServiceUnavailable: [503] during [POST] to [http:/
```

durationInMs:56
stateTransition:null
3{6}
circuitBreakerName:detailsForCustomerSupportApp
type:ERROR
creationTime:2023-01-17T20:55:45.760547300+05:30[Asia/Calcutta]
errorMessage:feign.FeignException$ServiceUnavailable: [503] during [POST] to [http:/
durationInMs:73
stateTransition:null
4{6}
circuitBreakerName:detailsForCustomerSupportApp
type:ERROR
creationTime:2023-01-17T20:55:47.519405+05:30[Asia/Calcutta]
errorMessage:feign.FeignException$ServiceUnavailable: [503] during [POST] to [http:/
durationInMs:44
stateTransition:null
5{6}
circuitBreakerName:detailsForCustomerSupportApp
type:FAILURE_RATE_EXCEEDED
creationTime:2023-01-17T20:55:47.525461900+05:30[Asia/Calcutta]
errorMessage:null
durationInMs:null
stateTransition:null
6{6}
circuitBreakerName:detailsForCustomerSupportApp
type:STATE_TRANSITION
creationTime:2023-01-17T20:55:47.542849200+05:30[Asia/Calcutta]
errorMessage:null
durationInMs:null
stateTransition:CLOSED_TO_OPEN
7{6}
circuitBreakerName:detailsForCustomerSupportApp
type:NOT_PERMITTED
creationTime:2023-01-17T20:55:49.637744+05:30[Asia/Calcutta]
errorMessage:null
durationInMs:null
stateTransition:null

- After 30 seconds, CB will half open allowing 3 requests. Hence error message will change. Then when threshold is reached error message will change again.

G) Implementing Fall back mechanism:
In above example when cards MS was down, we were not even able to receive response from MS which where working. Hence we will try to build a fall back mechanism in this lecture

- Add one more attribute in CB annotation in Accounts Microservice API:

@CircuitBreaker(name="detailsForCustomerSupportApp", **fallbackMethod = "myCustomerDetailsFallBack**")

This will redirect call to a secondary API in erroneous condition

- Implement fall back method:

```
private CustomerDetails myCustomerDetailsFallBack(Customer customer,
Throwable t) {
   Account accounts =
accountsRepository.findByCustomerId(customer.getCustomerId());
   List<Loans> loans = loansFeignClient.getLoansDetails(customer);
   CustomerDetails customerDetails = new CustomerDetails();
   customerDetails.setAccount(accounts);
   customerDetails.setLoans(loans);
   return customerDetails;

}
```

There are certain rules to follow while writing a fall back method:

- It should accept the exact same input as original method.
- Second argument must be Throwable, so that you can provide exception specific implementation
- In this example, we are simply not fetching data from Cards API, but you can write fallback method logic as per your business requirement, for example you can return default cards response or fetch data from cards cache etc.
- Please note that fall back method will be called only when there is error from cards microservice, otherwise we will receive reply from original method only.

3) Retry Pattern:
Sometimes micro service call can fail due to some network issue. In such cases we can make use of Retry Pattern where the call will be retried for a configurable no of times.

A) Important properties to be set while using Retry Pattern:

- For retry pattern we can configure the following values,
  - ✓ maxAttempts - The maximum number of attempts
  - ✓ waitDuration - A fixed wait duration between retry attempts
  - ✓ retryExceptions - Configures a list of Throwable classes that are recorded as a failure and thus are retried.
  - ✓ ignoreExceptions - Configures a list of Throwable classes that are ignored and thus are not retried.
- We can also define a fallback mechanism if the service call fails even after multiple retry attempts. Below is a sample configuration,

```
@Retry(name="detailsForCustomerSupportApp",fallbackMethod=
"myCustomerDetailsFallBack")
```

B) Implementing Retry Pattern:

- Add given below annotation in the method where you want to implement retry pattern:
  @Retry(name = "retryForCustomerDetails", fallbackMethod = "myCustomerDetailsFallBack")

- Add given below properties in application.properties file
  resilience4j.retry.configs.default.registerHealthIndicator= true
  resilience4j.retry.instances.retryForCustomerDetails.maxRetryAttempts=3
  resilience4j.retry.instances.retryForCustomerDetails.waitDuration=2000

- With retryAttempts of 3, call will be onvoked 3 times before invoking fallback method. Please note that in this case, original invocation will be called 3 times (i.e invocation on which annotation is applied)

4) Rate Limiter Pattern:
Rate Limiter pattern can be used for limiting the number of requests being sent to a micro service.

A) Pattern and configuration:

- This pattern protects micro service from DDOS attacks, cascading failure

- The rate limiter pattern will help to stop overloading the service with more calls more than it can consume in a given time. This is an imperative technique to prepare our API for high availability and reliability.
- This pattern protect APIs and service endpoints from harmful effects, such as denial of service, cascading failure.
- For rate limiter pattern we can configure the following values,
  - ✓ timeoutDuration - The default wait time a thread waits for a permission
  - ✓ limitForPeriod - The number of permissions available during one limit refresh period
  - ✓ limitRefreshPeriod - The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
- We can also define a fallback mechanism if the service call fails due to rate limiter configurations. Below is a sample configuration,

```
@RateLimiter(name="detailsForCustomerSupportApp",fallbackMethod=
"myCustomerDetailsFallBack")
```

- For example, if we set timeout duration to 1sec and limit for Period to 100 then it means the micro service can accept 100 requests in 1 sec, remaining request will wait in queue. If after 1 sec still pending 100s are not consumed then, pending requests will return with error “Too many requests”

B) Implementing Rate Limit Pattern:

- Add annotation in the API you want to protect
- Add given below properties in application.properties file:

```
resilience4j.ratelimiter.configs.default.registerHealthIndicator= true
resilience4j.ratelimiter.instances.sayHello.timeoutDuration=5000
resilience4j.ratelimiter.instances.sayHello.limitRefreshPeriod=5000
resilience4j.ratelimiter.instances.sayHello.limitForPeriod=1
```

With above setting we allow only 1 request within 5 seconds.

5) Bulk Head Pattern:

Bulk Head Pattern is similar to Bulk Head design in a ship. In a ship if one compartment is filled with water then entire ship is not sunk because we seal that compartment. Similar pattenr we can use in Micro service where we limit the number of resources to be used by a micro service, That way if a give nmicroservices crosses the threshold number of resources then it has to wait so that it does not eat up all resources. This will not block other API calls.

- For example, in our sample code we can apply this pattern on myCustomerDetails API as this is a complex API which calls other APIs.

ROUTING AND CROSS CUTTING CONCERNS

A) Challenges:

- How to route a request to a given end point based on url. For example, a client may want to invoke beta or stable version of a micro service
- How to handle cross cutting concerns like logging, auditing, security in micro service. We can give this responsibility to individual micro services, but then this will be a huge burden on individual developers, and they can also break the consistency. We can also use a common library but then, this will increase coupling.
  Hence, we need a common service which can take care of these issues
- How to monitor the inbound and outbound traffic and make it policy aware

The Spring cloud framework solves above challenges

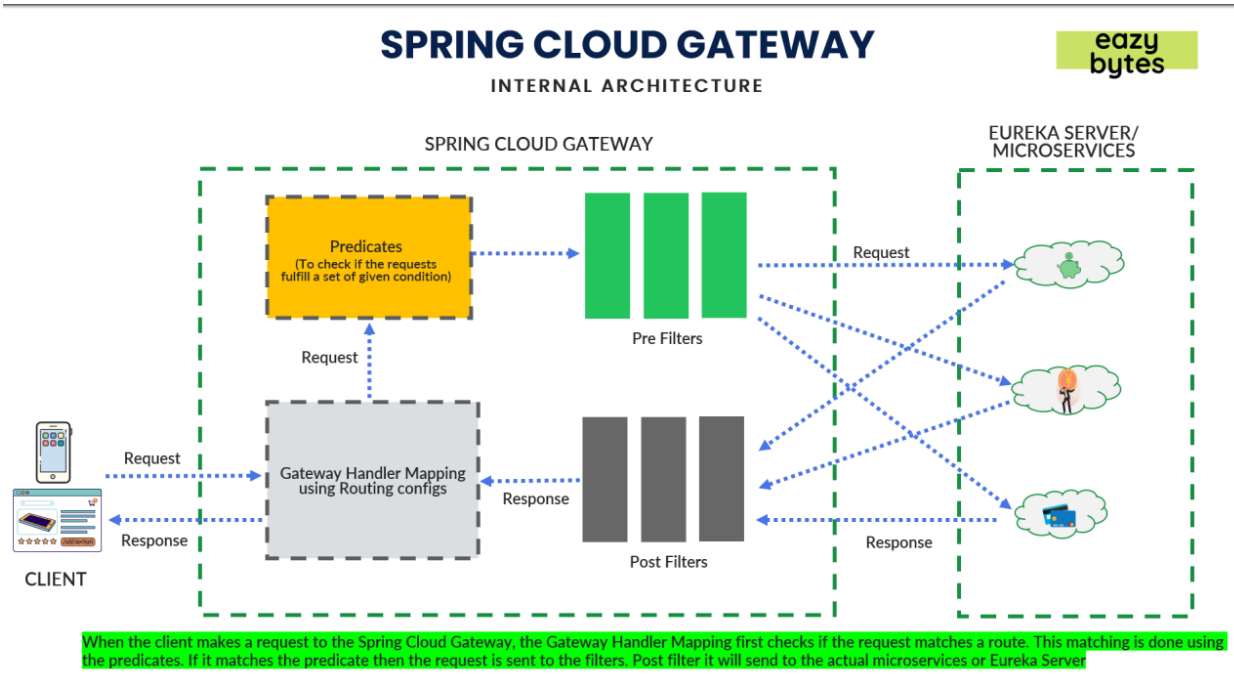
B) Spring Cloud Gateway:

- It's a framework used for building an API Gateway
- API gateway will seat between requestor and resource
- It will check and monitor incoming traffic, route request to an appropriate resource
- It can intercept and modify request and response based on custom requirements.
- One can argue that this is like Eureka server where also we route request by invoking Eureka Server.
- Eureka server acts as a service registry and monitors health of instances, it also expects you to invoke exact url.
- But an API Gateway also takes care of custom routing requirements, cross cutting concerns which cannot be achieved through Eureka Server
- It can also maintain sticky session. For example, a user A invokes a url. Based on URL API GW routes request to a certain destination/node/container. If the same user invokes request again then sticky session will ensure that same node will be returned to serve the request.

FOR ROUTING, CROSS CUTTING CONCERNS

- Spring Cloud Gateway is a library for building an API gateway, so it looks like any another Spring Boot application. If you're a Spring developer, you'll find it's very easy to get started with Spring Cloud Gateway with just a few lines of code.
- Spring Cloud Gateway is intended to sit between a requester and a resource that's being requested, where it intercepts, analyzes, and modifies every request. That means you can route requests based on their context. Did a request include a header indicating an API version? We can route that request to the appropriately versioned backend. Does the request require sticky sessions? The gateway can keep track of each user's session.
- Spring Cloud Gateway is replacement of Zuul for the following reasons and advantages,
  - Spring Cloud Gateway is the preferred API gateway implementation from the Spring Cloud Team. It's built on Spring 5, Reactor, and Spring WebFlux. Not only that, it also includes circuit breaker integration, service discovery with Eureka
  - ✓ Spring Cloud Gateway is non-blocking in nature. Though later Zuul 2 also supports it, but still Spring Cloud Gateway has an edge here
  - ✓ Spring Cloud Gateway has a superior performance compared to that of Zuul.

C) Spring Cloud GW Internal Architecture:



- A client sends a request to our Micro service via API GW
- Request is routed to Gateway Handler Mapping which will check if the incoming url has a corresponding route mapping. The purpose of Gateway Handler Mapping is to resolve the actual end point address (Eureka Server or actual Micro service).
- If any custom routing info is not configured then, Spring Cloud GW is smart enough to resolve destination to Eureka Server or Micro service as it has got integration with Eureka Server
- After that Predicates are executed which are similar to Java 8 Predicates.
  - ➢ The can check is incoming request is authorized or not

  - The incoming request contains a mandatory param or not
  - If requesting IP is allowed or not etc

-
- After this Pre Filters are executed. This is the place where we can take care of cross cutting concerns like logging, tracing, auditing
- Now request is sent to Eureka Server or Micro service instance
- Once Micro Server instance process the request, the response is sent to Post Filters. Here again we can take care of cross cutting concerns like adding a header in response or calculating the actual time of request processing
- The response is then finally sent via Gateway Handler Mapping

D) Building Spring Cloud GW Service:

- Create Spring Starter gatewayserver project in spring.io with Actuator, Gateway, Eureka Server, Config Server, Dev Tools as dependency
- Add given below annotation in main class:
  @EnableEurekaClient
  This is required so that Gateway Server MS will register itself in Eureka Server (As internal MS can also invoked other MS via Gateway Server).
- Add given below entries in application.properties file:

  spring.application.name=gatewayserver
  spring.config.import=optional:configserver:http://localhost:8071/
  management.endpoints.web.exposure.include=*
  ## Configuring info endpoint
  info.app.name=Gateway Server Microservice
  info.app.description=Eazy Bank Gateway Server Application
  info.app.version=1.0.0
  management.info.env.enabled = true
  management.endpoint.gateway.enabled=true
  spring.cloud.gateway.discovery.locator.enabled=true
  spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true

  logging.level.com.eaztbytes.gatewayserver: DEBUG

- Create new configuration file gatewayserver.properties file:
  server.port=8072
  eureka.instance.preferIpAddress = true
  eureka.client.registerWithEureka = true
  eureka.client.fetchRegistry = true
  eureka.client.serviceUrl.defaultZone = http://localhost:8070/eureka/

- Invoke the url where Gateway Server is up and append the url with actuator. Here find the url for gateway so that you check what are the default routing information exposed by Gateway Server.
  It will look like: http://localhost:8072/actuator/gateway/routes

- Hit the url found in previous step. Here you will find that Gateway server loads all registry information from Eureka Server on start up.
- Finally invoke the Micro service using Gateway Server:
  http://localhost:<gateway-server-port>/accounts/myCustomerDetails

Upon invoking above url Gateway Server will locate Micro service instance from Eureka registry and invoke the service

Please note that here we have not used any custom routing, pre-filter or post-filter

C) Implementing Custom Routing:

- Suppose your end customer want to invoke request using a company name in input request. For example, they can invoke request using given below url:

  http://localhost:<gateway-server-port>/<company-name>accounts/myCustomerDetails

In above case we need to remove <company-name> from input url to locate actual micro service from Eureka Server

- Create a new method to inject a spring bean of type RouteLocator

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder){

    return builder.routes()
            .route(p -> p
                    .path("/eazybank/accounts/**")
                    .filters(f ->
f.rewritePath("/eazybank/accounts/(?<segment>.*)","/${segment}")
                            .addResponseHeader("X-
Response-Time",new Date().toString()))
                    .uri("lb://ACCOUNTS")).
            route(p -> p
                    .path("/eazybank/loans/**")
                    .filters(f ->
f.rewritePath("/eazybank/loans/(?<segment>.*)","/${segment}")
                            .addResponseHeader("X-
Response-Time",new Date().toString()))
                    .uri("lb://LOANS")).
```

```
                route(p -> p
                        .path("/eazybank/cards/**")
                        .filters(f ->
f.rewritePath("/eazybank/cards/(?<segment>.*)","/${segment}")
                                .addResponseHeader("X-
Response-Time",new Date().toString()))
                        .uri("lb://CARDS")).build();


        }
```

D) Implementing Custom Filter based Tracing

- There can be requirement to trace incoming requests
- Intention is to assign trace ID to each incoming request.
- Then we can return the same trace ID in response Header
- That way if any request fails then client can share the specific trace id from response header to us.
- By tracing the same trace ID in logs, we can check till where invocation was successful and where it failed.

DISTRIBUTED TRACING AND LOG AGGREGATION

A) Challenges:

- How to trace a specific request in a Micro Service based distributed environment. For example, a request initiated by client may span multiple microservices distributed across multiple containers/nodes/azs. How to trach a single request in such cases
- How to aggregate logs? Take previous example where a request span multiple micro services. Each micro service will generate its own log. How to aggregate logs from multiple micro services in a single place.
  When above two problems are solved then we can also debug issues easily.

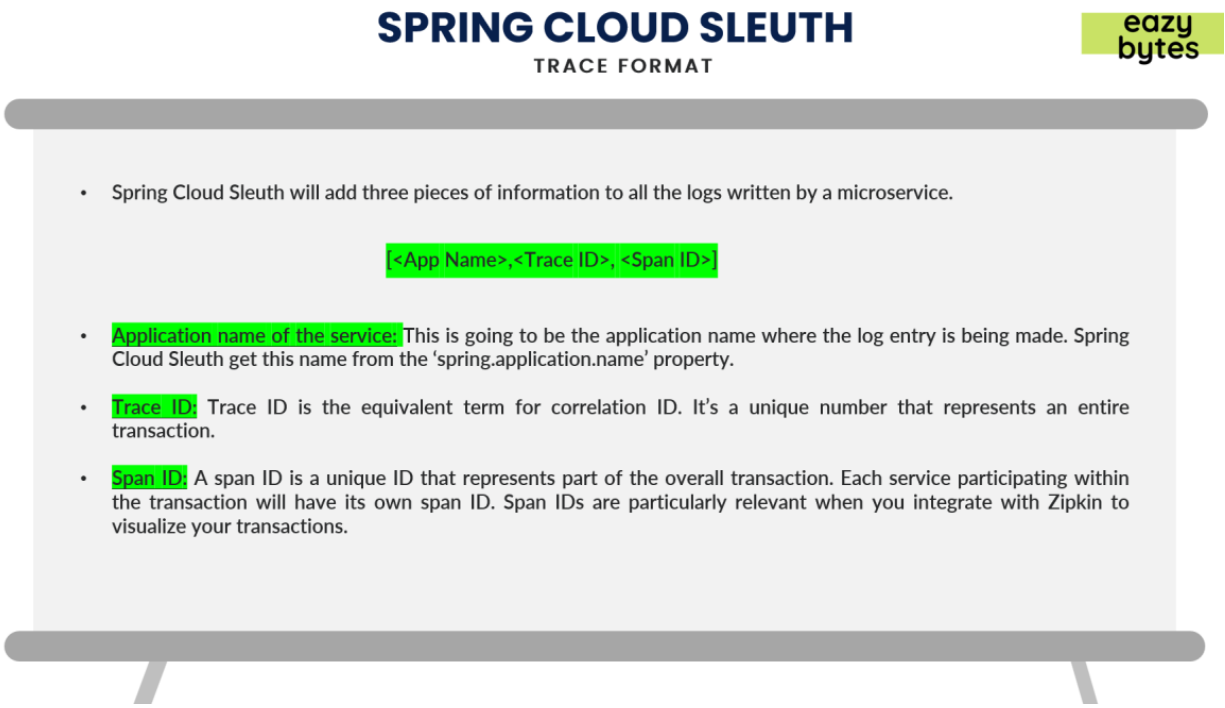
  Spring cloud provided APIs to solve above issues.

B) Spring Cloud Support for Distributed tracing and Log Aggregation

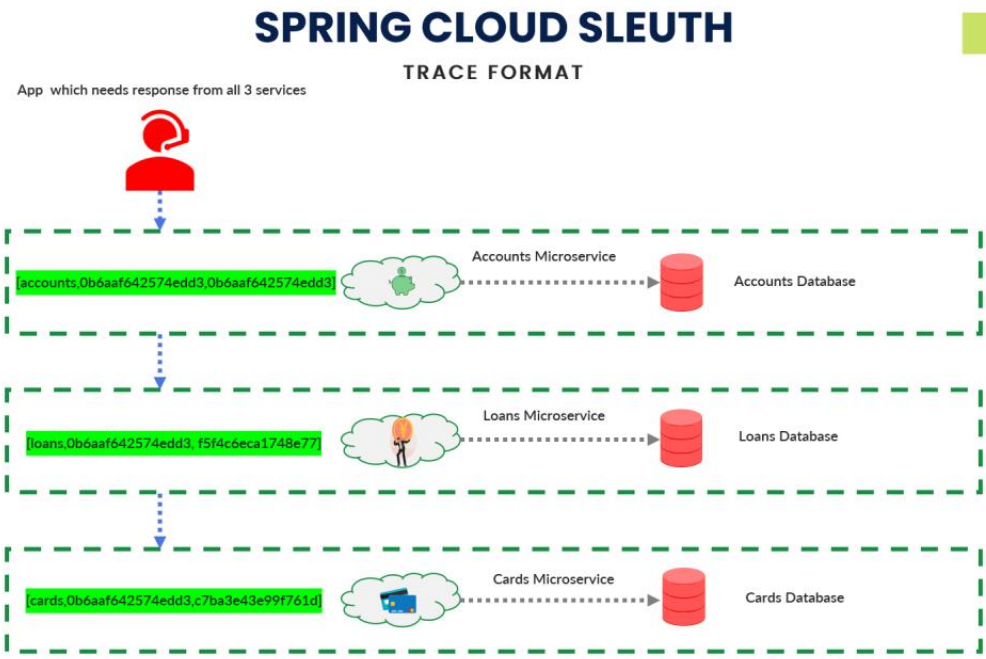
- Spring cloud sleuth will add a trace id and span id for each log statement. Thereby we can search by trace id or span id.
  It does so by intercepting each incoming request and generating id for each call and appending that id across multiple calls.
- Zipkin performs the task of aggregating logs from multiple microservice location into a single centralized location.
-

It also breaks down a transaction or a call into individual components to identify hot spots, performance issues etc.

C) Sleuth Deep Dive:



-

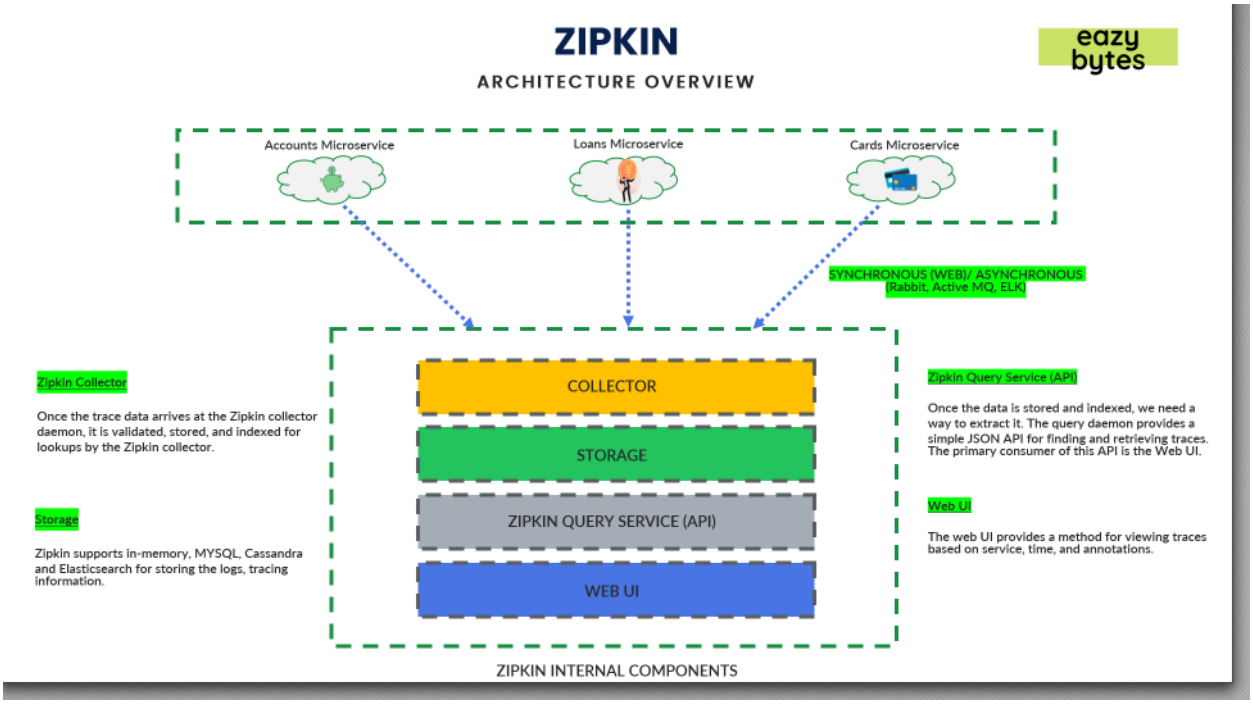


-

We can observe that trace ID remains same, but span id changes when service changes. Please note that for first microservice span id and trace id are same.

D) Zipkin Deep Dive:

- Each individual micro service will write logs information to a centralized zip kin server. For that a small configuration regarding zipkin server location will be required, rest will be taken by spring cloud framework.
- Log information can be pushed to zip kin server in a synchronous manner, but that might be slow
- Log information can also be pushed to zip kin server in an asynchronous manner by pushing log information to Rabbit MQ or a JMS Queue. Zip kin can then be configured to listen from that queue.
- One can also write log information to ELK or Splunk but in this case we will discuss about zip kin server.
- Zip kin server has four components: Collector, Storage, Zip Kin Query Service API, Web UI



In this tutorial we will discuss In Memory Storage of logs in Zip Kin.

E) Implementing Distributed Tracing:

- Add given below sleuth dependency in all micro services

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

- Start all the micro services

- Invoke Account's myCustomerDetailsAPI which also invoked Loans and Card API

- Now check logs for each individual micro service in console

Accounts Microservice:
2023-04-02 00:20:45.690 INFO [**accounts**,**69ffeccad0ba7b9f**,**69ffeccad0ba7b9f**] 19392 --- [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'

Since accounts is the first API to get called hence span is and trace id are same

Cards Microservice:
2023-04-02 00:20:46.837 INFO [**cards**,**69ffeccad0ba7b9f**,**cae7fded484b9699**] 8824 --- [nio-9001-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'

Loans Microservice:
2023-04-02 00:20:46.420 INFO [**loans**,**69ffeccad0ba7b9f**,**0dc363f6639ff543**] 17004 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'

F) Implementing Log Aggregation:
For implementing Log Aggregation using Zip kin, we will start zip kin server (on a docker container) and then we will configure each individual microservice to locate zip kin server. Thereby logs from each micro service will be written to zip kin server.

- Add dependency for zip kin in pom.xml

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

- Add given section to start zipkin in a docker container in docker compose file:

```
zipkin:
  image: openzipkin/zipkin
  deploy:
    resources:
      limits:
        memory: 700m
  ports:
    - "9411:9411"
  networks:
    - hitesh791-network
```

- Add dependency for zipkin server in each micro service:

```
-   depends_on:
      - zipkin
```

- Add env variable for zip kin server location:

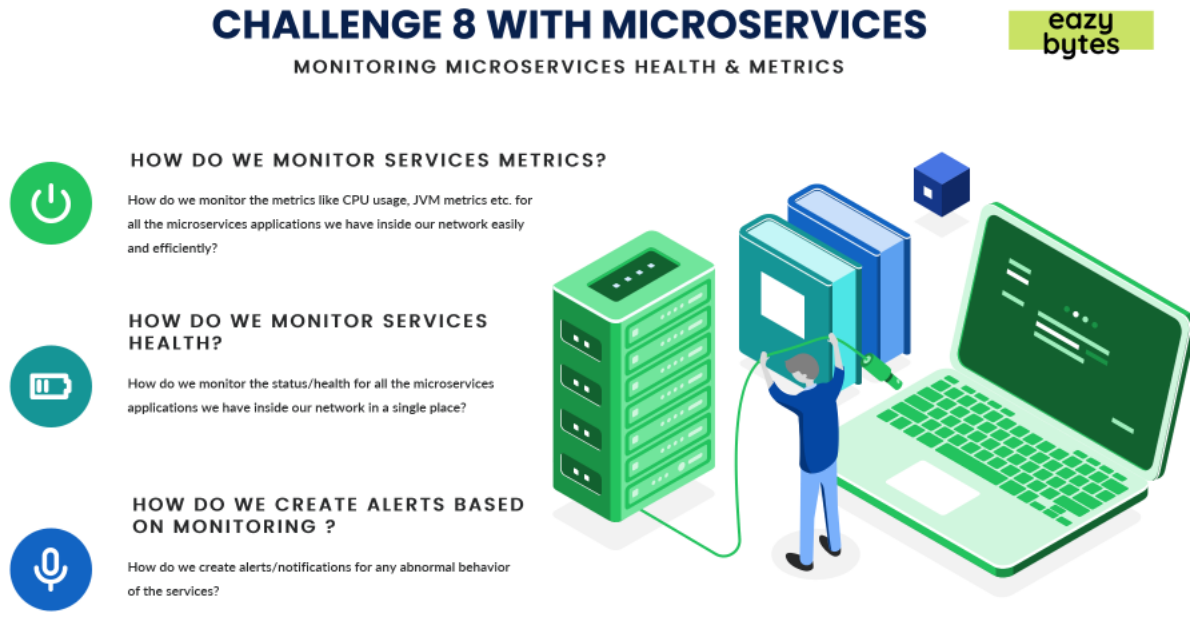
```
SPRING_ZIPKIN_BASEURL: http://zipkin:9411/
```

- Now restart all micro services, create docker images and run docker compose file

MICROSERVICE MONITORING

We can have 100s of microservices running in a distributed environment. In such case we need a mechanism to monitor health, statistics, and metrics of individual microservices.

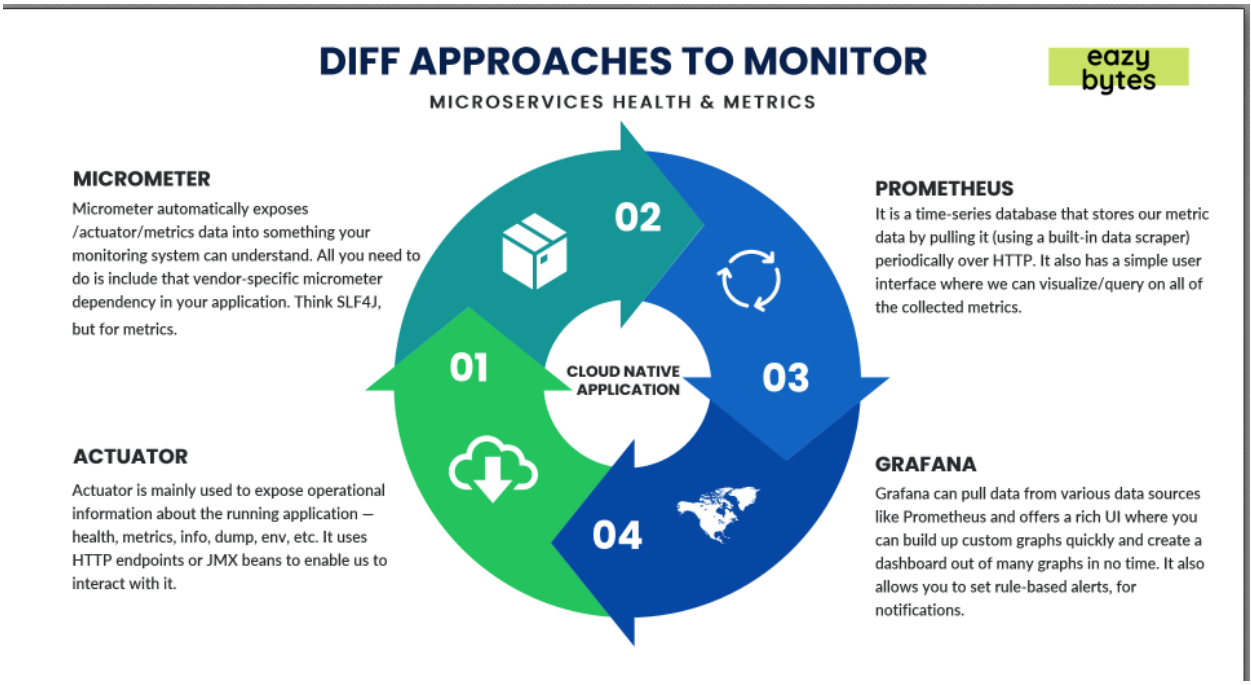
A) Challenge:



B) Approaches:

- Actuator:
  - Actuator by default exposes end points to get information regarding a running micro service like health, metrics, dump etc.
  - This is a very basic approach.
  - In case we have multiple instances of microservices then we have to visit url ofr each instance to get information from actuator. This will be tedious task.
- Centralized Framework Approach:
  - There should be a centralized app from where we can get all monitoring related information for all microservices/instances.
  - This is where Micrometer, Prometheus and Grafana comes into play.
  - But this should not be responsibility of individual microservice instances to send information to Prometheus as this will be performance issue.
  - Hence its Prometheus responsibility to collect information from individual microservices by calling actuator end point url exposed by microservice instances.
  - But data coming from actuator needs to be converted into format that Prometheus understands. This is done by Micrometer.

- Micrometer is a generic framework for data conversion. It can also convert data into ELK format. You just need to add relevant micrometer plugin.
- The interval within which Prometheus pulls data from actuator via Micrometer is configurable.
- But Prometheus has limited functionality to offer in its UI. This is where Grafana comes into play.
- Grafana pulls data from Prometheus to render a very rich monitoring UI. Here alerts can also be configured. Grafana has lot to offer in this context.



C) Setting Up Micrometer in Micro services:

- Add given below dependency in microservices:

```
-   <dependency>
       <groupId>io.micrometer</groupId>
       <artifactId>micrometer-core</artifactId>
    </dependency>
    <dependency>
       <groupId>io.micrometer</groupId>
       <artifactId>micrometer-registry-prometheus</artifactId>
    </dependency>
```

First entry is for using micrometer libraries.
Second entry is for using Prometheus plugin for micrometer. Similarly, plugin can be used for ELK and AWS cloud watch.

- One can also create custom metric to be sent to Prometheus via micrometer. For this, ass given below dependency in pom.xml:

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

- For custom metric add given below @Bean entry in AccountsApplication class. This will inject the bean in spring context.

```
@Bean
public TimedAspect timedAspect(MeterRegistry registry){
    return new TimedAspect(registry);
}
```

This will enable aspect to derive time spent in a service execution.

Add given below annotation in the method in which you want to calculate the time:

```
@PostMapping("/myAccount")
@Timed(value = "getAccountsDetail.time", description = "Time taken to return
account details")
public Account getAccountsDetail(@RequestBody Customer customer){
   Account account =
accountsRepository.findByCustomerId(customer.getCustomerId());
   if(account!=null){
      return account;
   }
   return null;
}
```

- Once you bring micro services up then actuator will expose a new url where actuator metric data will be available in Prometheus format. Prometheus will invoke this url in regular intervals to collect and collate data

For example, for accounts micro service, the url will be like: http://localhost:8080/actuator/prometheus

D) Setting up Prometheus:
Now its time to set up Prometheus so that it can collect data from actuator.

- Note that we need to set up Prometheus in a way that it should be able to pull data from each micro service instance.
- Hence each instance should know information regarding Prometheus details.
- For this create a file Prometheus.yml with given below content:

```
global:
            scrape_interval:     5s # Set the scrape interval to every 5 seconds.
            evaluation_interval: 5s # Evaluate rules every 5 seconds.
          scrape_configs:
            - job_name: 'accounts'
              metrics_path: '/actuator/prometheus'
              static_configs:
              - targets: ['accounts:8080']
            - job_name: 'loans'
              metrics_path: '/actuator/prometheus'
              static_configs:
              - targets: ['loans:8090']
            - job_name: 'cards'
              metrics_path: '/actuator/prometheus'
              static_configs:
              - targets: ['cards:9000']
```

Under scrap config we define mircro services whose instance information we need to read via actuator url. Metric Path refers to path exposed by Actuator. Static config defines end point where micro service instance is running. In above example we have specified that Prometheus should collect actuator info for accounts micro service instance running in port 8080. For other instances we need to provide comma separated values in target section. Please note that we specify target in the form accounts:8080 as in docker ompose environment we identify location of a micro service via service name rather than actual ip.

- Add given below changes in docker compose file:

```
prometheus:
                image: prom/prometheus:latest
                ports:
                   - "9090:9090"
                volumes:
                 - ./prometheus.yml:/etc/prometheus/prometheus.yml
                networks:
                 - eazybank
```

Here we are pulling Prometheus image from docker hub. In the volume section we are instructing to copy Prometheus.yml file in location /ect/Prometheus/promethus.yml in container. This is required as when Prometheus container is launched and run then it will look for Prometheus configuration information in folder `/etc/prometheus/prometheus.yml`

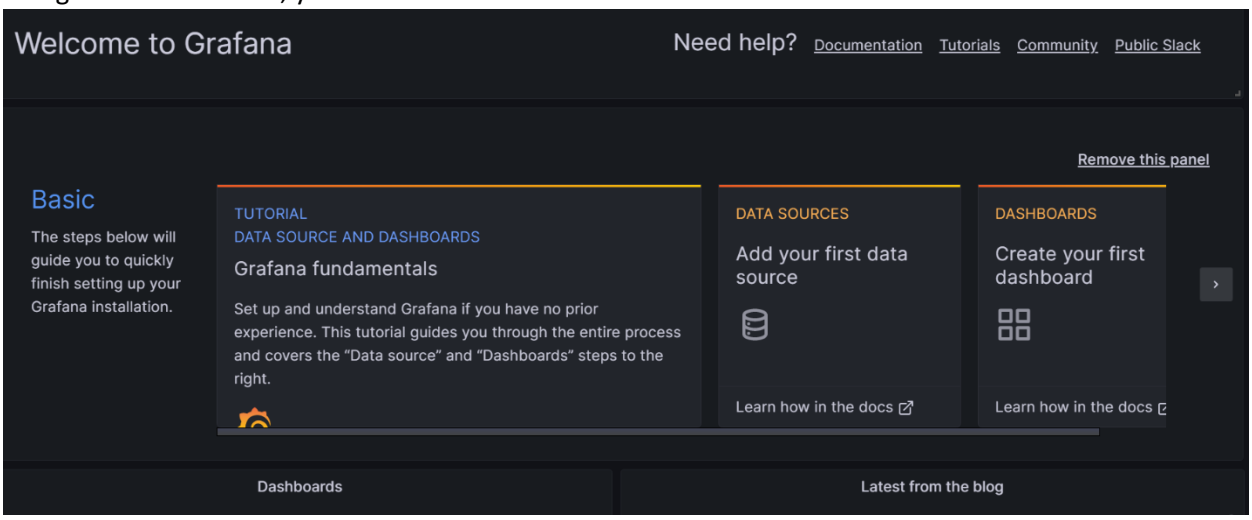
- Keep docker compose file and Prometheus.yml file in same location.
- Execute command docker-compose up
- Navigate to url: http://<ip-address>:<port>/actuator/prometheus
  Here you will not see our custom metric yet as we have not hit the getAccountDetailsAPI yet. Hit the API then you will the metric for the same in above url
- Navigate to Prometheus url: http://<ip>:<port>:9090.
  Here you can see all metrics related to micro services instances you configured in Prometheus.yml.

E) Setting Up Grafana:
If the monitoring feature offered by Prometheus is not enough then, you can configure Grafana. Grafana will pull data from Prometheus to create more interactive dashboards and UIs.

- Add given below image entry for Grafana in docker compose file:

```
grafana:
  image: "grafana/grafana:latest"
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=password
  networks:
    - hitesh791-network
  depends_on:
    - prometheus
```

- Execute docker compose
- Navigate to Grafana url, you will see screen like below:



- Add Prometheus as data source, if configured correctly then your connection test will be successfull

- You can create custom dashboards, but some inbuilt dash boards are also available. For that you can visit Grafana web site and import dashboards from there using url

CONTAINER ORCHESTRATION

A) Challenge:
- Auto Deployment
- Deployment of newer version without downtime
- Automatic Scale Up, Scale Down, Healing
- Management in a complex cluster

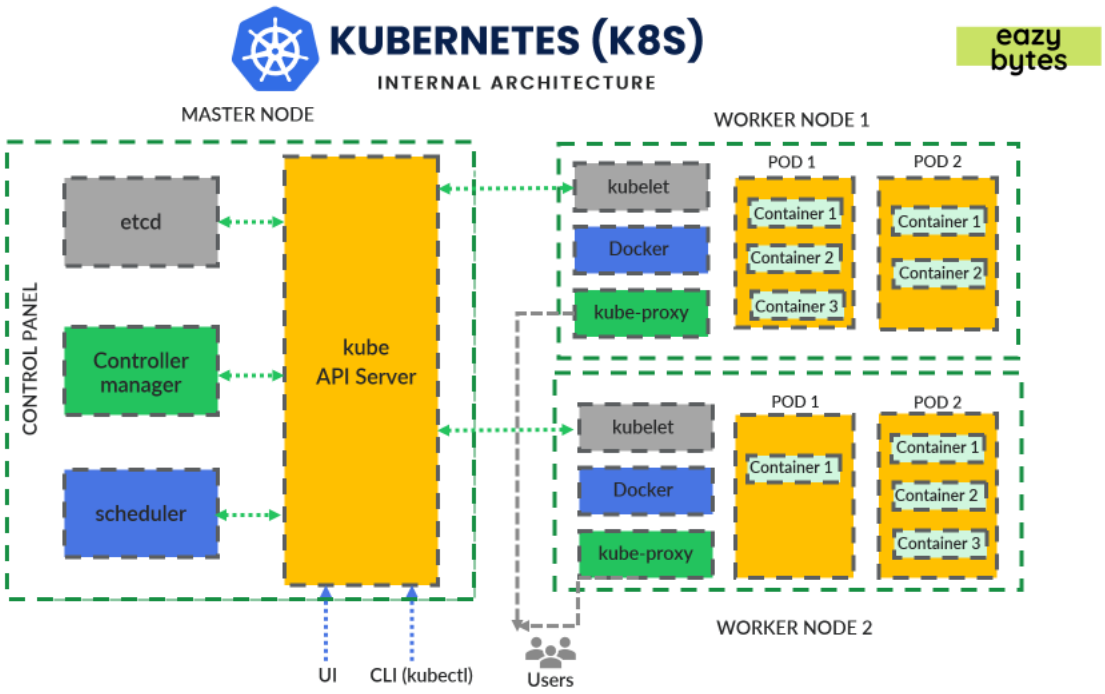
KUBERNETES

K8s is an orchestration framework used for maintaining, scaling, self-healing, deployment of containers.

It provides:

- Service Discovery and Load Balancing
- Storage Orchestration
- Automated Roll outs and rollbacks
- Automates bin packing
- Self-healing
- Secret and configuration management (No config server required)
- It cloud neutral. It can be run and deployed on On Prem. AWS, GCP and Azure also provides support for K8s

A) Deep Dive of Kubernetes Architecture:



- Just like in any distributed computing environment, K8s also follows a clustered architecture.
- It consists of Master Node and Worker Nodes
- There can be any number of Worker Nodes like 10, 50, 100 or 1000. You can have one or more Master Nodes, depending in number of Worker Nodes
- Worker Nodes is where containers are deployed and run.
- Master node makes sure that Worker Nodes are working properly.
- Maser Node has 4 important components:
  - ➢ kube API Server:
    - Its like a set of rest services which represents all operations that can be done by master node.
    - Anyone who wants to interact with cluster then that is facilitated through kube API Server.
    - So kube API is like a Gateway to cluster.
    - It also acts like a Gate Keeper for the cluster so that only authenticated users are interacting with the cluster.
    - There are two ways to interact with kube API Server: UI and (CLI)kubectl.
    - So, to interact with K8s cluster you only tell Master Node what to do. Request can be like adding a new container, scaling up scale down containers, adding more container instances, self-healing etc. Eventually that request is served by kube API Server.

  - ➢ scheduler:

- Upon a receiving a request, kube API server forwards that request to scheduler.
- A request can be for example to deploy a new microservice. So for example, a user can initiate a request to deploy accounts micro service with 3 replicas
- Scheduler will receive a request like this user want to deploy three containers for this docker image.
- Scheduler will then perform internal calculation and check which Worker Node has less load.
- Upon choosing a Worker node, deployment of micro service container will be scheduled in chosen worker node.
- Worker Node will then take instruction from scheduler to deploy containers in POD.

➢ Controller manager:
- Once can also give deployment instruction like I want three instances of Accounts Micro service always.
- So, it will be job of controller manager to ensure that three instances(desired) of micro service are always available. It will accomplish this through health checks.
- So, there is a desired state and a current state. Controller Manager ensures that desired state is same as current state.

➢ etcd:
- Its like Namenode in Hadoop cluster
- It stores meta data information
- Its brain of your cluster in the form of a database (where info is stored in key value form)
- It has information like number of worker nodes, location of worker nodes, number of PODS, number of microservices containers (desired)
- Scheduler when receives deployment instruction then it writes all relevant information in etcd (like number of desired container for a micro service)
- Controller Manager queries etcd to get desired state and then compare it with current state.

- All these 4 components together are called as Control Panel
- Since master node only performs duty of managing worker nodes and containers, it requires very less server capacity.
- But Worker Nodes are doing the heavy duty job, hence they require more server capacity.
- Worker Node has three important components:
  - ➢ kubelet:
    - Its an agent running on each Worker node
    - Kubelet communicates with Master Node to get instructions from Scheduler

- Once we raise a deployment instruction via kube API Server, then it is forwarded to Scheduler
- Scheduler will then select Worker Node based on lad and other parameters.
- Once it chooses a Worker Node then it pass on instruction to Worker Node via kubelet to deploy container inside a POD

➢ Docker:
- Since our container will run on docker, hence each Worker Node will have docker installed on it.

➢ kube-proxy:

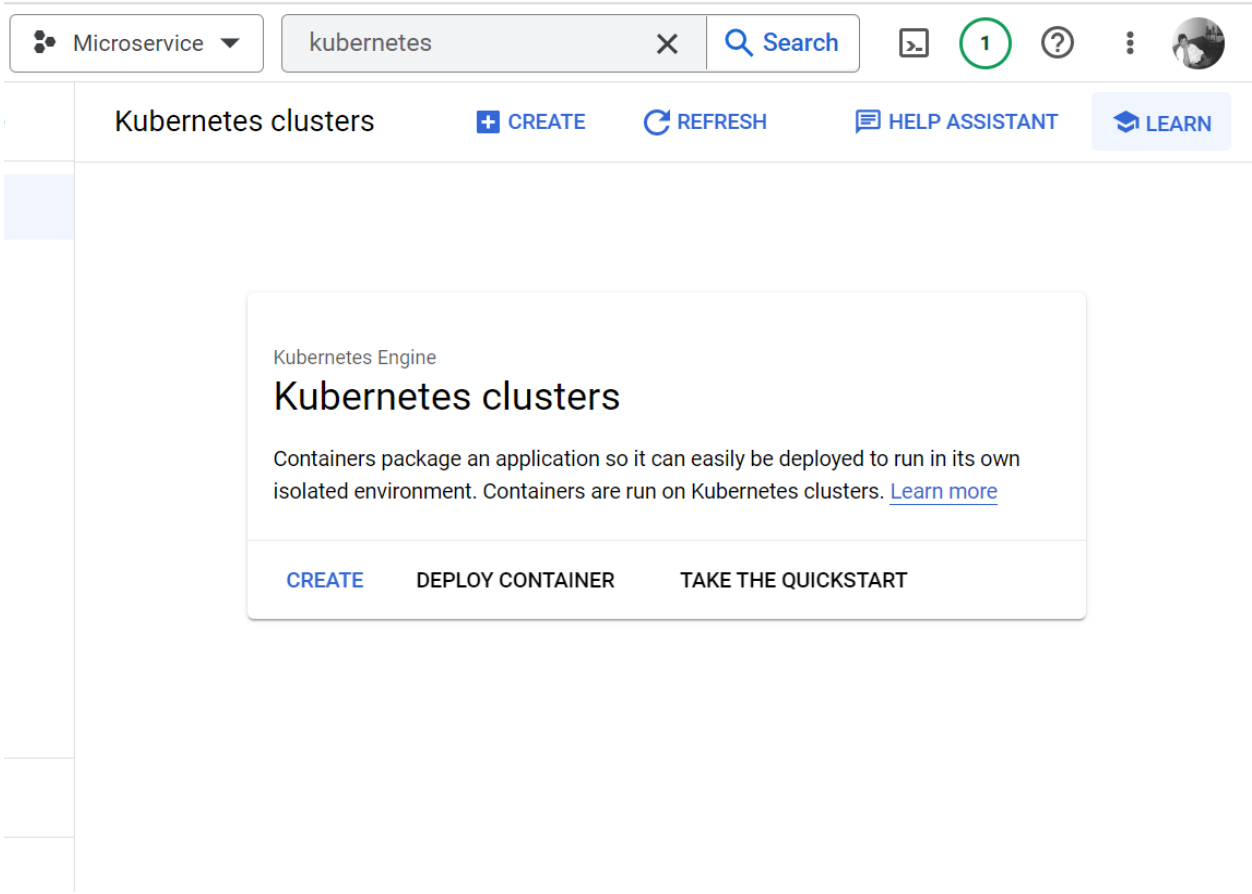
- Via kube-proxy you can expose your container end point urls:
- You make your services private or public
- It will also help in setting firewall settings
- End user willing to invoke service of a container has to invoke via kube-proxy.

➢ POD:
- POD is a smallest deployment component used to deploy and run containers. Just like container is a smallest unit of deployment in docker.
- When a deployment instruction is sent via kube API Server then that request is forwarded to Scehduler
- Scheduler will then choose a Worker Node based on load and other parameters
- Once a Worker Node is chosen then, scheduler will pass instruction to kubelet of that Worker Node.
- Kubelet will then create a new POD where the container will be deployed and run.
- POD can be treated like a mini version of Worker Node having its own memory, RAM, CPU. Inside POD containers run.
- Usually in POD singe container is run. Its not recommended to run multiple containers inside a POD.
- However, you can run helper containers along with main container inside a single POD.
- Once POD is created it will get an ip address assigned by pod networking solution and port to interact with containers. This information will be available via kube-proxy which can be used by end users to invoke container logic/business logic.

- What we call Pseudo Distribution in Hadoop is called as mini-kube in K8s.
- GCP's K8s service comes in free tier unlike in AWS and Azure.

B) Creating a K8s cluster in GCP:

- Create an account in GCP
- Create a new project
- Under the new project search for Kubernetes Service. Below page will be rendered:



- Click on Create to create a new K8s cluster, keeping default setting.
- Choose Auto pilot option
- By default, cluster will be created with 3 nodes
- Download and install GCP CLI by following instructions from cloud.google.com/sdk

C) Creating cluster and creating connection with it:
Once Cluster is ready we can explore it and try to connect with it using Google Cloud SDK

- Click on cluster name link to get basic details of cluster
- Click on nodes tab to see the nodes details:

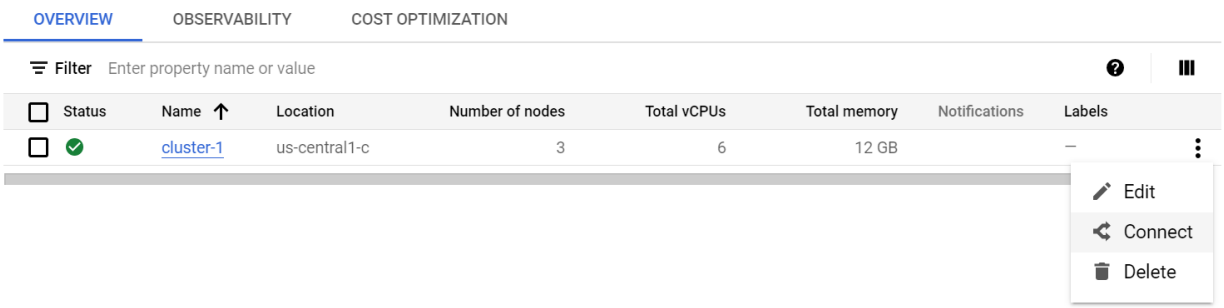
Name	Status	CPU requested	CPU allocatable	Memory requested	Memory allocatable
gke-cluster-1-default-pool-d7b46a76-1tnd	Ready	276 mCPU	940 mCPU	487.59 MB	2.95 GB
gke-cluster-1-default-pool-d7b46a76-1z0g	Ready	528 mCPU	940 mCPU	534.29 MB	2.95 GB
gke-cluster-1-default-pool-d7b46a76-g3hl	Ready	488 mCPU	940 mCPU	492.83 MB	2.95 GB

One of the nodes is master and remaining two are worker nodes

- Click on the node, and you will see given below details:
  - Some monitoring stats of the node
  - Log information in node
  - Events happening inside node
  - Default PODS created by GKE to accomplish above activities

So, in essence by default we have some monitoring ang logging enabled in out cluster.

- To enable logging and monitoring for any service deployed cluster we need to enable given below services:
  - API and Services -> Cloud Logging API
  - API and Services -> Stack driver API
  - API and Services -> Stack Driver Monitoring API
- Open your Kubernetes cluster. Click on ellipsis and choose option of connect:



- Upon clicking in Connect, you will get a command to connect to cluster
- Open Google cloud SDK CLI and execute that command:
  If you face error like given below:

  CRITICAL: ACTION REQUIRED: gke-gcloud-auth-plugin, which is needed for continued use of kubectl, was not found or is not executable. Install gke-gcloud-auth-plugin for use with kubectl by following https://cloud.google.com/blog/products/containers-kubernetes/kubectl-auth-changes-in-gke

  Then, execute given below two commands:

  a) gcloud components install kubectl
  b) gcloud components install gke-gcloud-auth-plugin

Now once again establish connection with cluster. Upon successful connection you will see given below message:

Fetching cluster endpoint and auth data.

kubeconfig entry generated for cluster-1.

- Execute command **kubectl get nodes**
  This will give you nodes details of your cluster

- Execute command **kubectl get pods**
  This will give you details of pods running in your cluster. Since there are no pods running in cluster therefore it will not give any details. However, internal pods are already running but they are used by GKE framework internally.

- Execute command **kubectl get deployment**
  This will give you details of any deployment done inside your K8s cluster

- Execute command kubectl get all
  This will give you details regarding services running inside your cluster. Right now there will be one default framework service running inside your cluster. Service is very important concept in K8s. By service we expose nodes and application deployed in K8s to outside world. Kubeproxy internally creates services to expose details of the service.

D) Deploying Microservices in K8s cluster
It's now time to deploy our micro services in K8s cluster

- We will create yaml files to provide deployment information to K8s regarding which docker image to use, no of replications etc.
- We will also create a yaml file by which we will establish connection between given set of micro services. For example account needs to know location of config server and eureka server. Zip kin needs ot know location of microservice to import log details.
- Given below is example of zip kin yaml file:

```
apiVersion:
apps/v1
              kind: Deployment
              metadata:
                name: zipkin-deployment
                labels:
                  app: zipkin
              spec:
                replicas: 1
                selector:
                  matchLabels:
                    app: zipkin
                template:
                  metadata:
                    labels:
                      app: zipkin
                  spec:
                    containers:
                    - name: zipkin
                      image: openzipkin/zipkin
                      ports:
                      - containerPort: 9411
              ---
              apiVersion: v1
              kind: Service
              metadata:
                name: zipkin-service
              spec:
                selector:
                  app: zipkin
                type: LoadBalancer
                ports:
                  - protocol: TCP
```

```
    port: 9411
    targetPort: 9411
```

- app version refers to kube server api version
- kind refers to whether its Deployment or service. In Deployment we give deployment information
- metadata gives more information like name. here we also specify label. This label name can be referred in other sections of yaml
- spec is very important section where we give information like no of replicas, container name, image to be pulled from docker hub, container port
- three hyphens --- indicates to Kubernetes that we have two different kind of yaml files in a single file
- Second section gives service information. Here we give details of how service will be exposed to outside world.
- We specify name, with this name we identify the service in other yaml configuration
- Port refer to port with which service will be exposed to outside world and target port is the internal port for microservice.
- Type Load Balancer will ensure that K8s engine will automatically take care of Load Balancing for multiple replicas/instance of the microservice.
- Target port maps to container port. With port we interact with micro service externally which redirect to actual container port using value of target port

- Lets take example of config server yaml file:

```
apiVersion:
apps/v1
             kind: Deployment
             metadata:
               name: configserver-deployment
               labels:
                 app: configserver
             spec:
               replicas: 1
               selector:
                 matchLabels:
                   app: configserver
               template:
                 metadata:
                   labels:
                     app: configserver
                 spec:
                   containers:
```

```
      - name: configserver
        image: eazybytes/configserver:latest
        ports:
        - containerPort: 8071
        env:
        - name: MANAGEMENT_ZIPKIN_TRACING_ENDPOINT
          valueFrom:
            configMapKeyRef:
              name: eazybank-configmap
              key: MANAGEMENT_ZIPKIN_TRACING_ENDPOINT
        - name: SPRING_PROFILES_ACTIVE
          valueFrom:
            configMapKeyRef:
              name: eazybank-configmap
              key: SPRING_PROFILES_ACTIVE
---
apiVersion: v1
kind: Service
metadata:
  name: configserver-service
spec:
  selector:
    app: configserver
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8071
      targetPort: 8071
```

- Some of the details are same as before.
- We have added a new section “env” which is used for defining environment variables (the same that we used in docker compose file). Config server needs to know location of zip kin server which is defined in env section. We are referring location by reading configmap yaml file (defined in next section). We give name of config name file and key name to get variable value. Similarly we have also specified profile to be used by config server in env section.
- Please note that other micro services also need to know location of config server, hence we have defined service section. Remember in K8s cluster, services are referred by their name as defined in service section in yaml file.

- Content of configmaps.yaml:
  Given below is content of configmap.yaml:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: bankacc-configmap
data:
  SPRING_ZIPKIN_BASEURL: http://zipkin-service:9411/
  #MANAGEMENT_ZIPKIN_TRACING_ENDPOINT: http://zipkin-
service:9411/api/v2/spans
  SPRING_PROFILES_ACTIVE: default
  SPRING_CONFIG_IMPORT: configserver:http://configserver-service:8071/
  EUREKA_CLIENT_SERVICEURL_DEFAULTZONE: http://eurekaserver-
service:8070/eureka/
```

You can see that the information that we provided in docker-compose file is now in configmaps.yaml.
However, our config server is still loading configuration information via git hub config repo.

- Uploading configuration information in K8s cluster:
  - Connect to cluster using gcloud CLI
  - Execute command **kubectl get config map** to get information of existing config maps
    Please note that there are already some default config maps which are used internally by GKE
  - Navigate to directory where config map file is located in gcloud terminal
  - Execute command **kubectl apply -f <config-file-name>**
  - Navigate to GCP console to check if config map is created by clicking on Storage and ConfigMaps links on right pane
  - For deleting config map execute command: **kubectl delete configmap <config-map-name>**

    Please note that you can also create config maps using kubectl shell provided in GCP console. In GCP console you can also view YAML based config map in K8s cluster. If you want to use same config map then we can use this YAML.

    Config Maps are a good replacement for Config server. But if we have large amount of configuration information then, config server is a better approach.

- Deploying Micro service in K8s

  - Microservice will be deployed in a specific sequence in our example use case. Config Map, Zip kin, Config Server, Eureka Server and then Business Micro service
  - Connect to cluster in gcloud CLI

- Navigate to directory where yaml files are located in gcloud terminal
- Execute command **kubectl apply -f <yaml-file-name>** to deploy microservice in K8s cluster.
  This command will deploy micro service in K8s cluster. K8s engine will create POD where instance of the micro service will be running
- Execute command **kubectl get pods**.

```
C:\Users\z0034ydd\MicroService-Training-Latest\accounts\kubernetes>kubectl get
NAME                                    READY   STATUS    RESTARTS      AGE
accounts-deployment-65454fb8c-7hpv5     1/1     Running   2 (72s ago)   98s
configserver-deployment-c664d5577-kfgx7 1/1     Running   0             119s
eurekaserver-deployment-5d7f4f579c-fc59f 1/1    Running   2 (76s ago)   108s
zipkin-deployment-6d588859dc-pwgnj      1/1     Running   0             2m14
```

As we can see that instances are up and running. In 1/1 first 1 indicates current replica and second 1 indicates desired replicas

- Execute command **kubectl get services:**

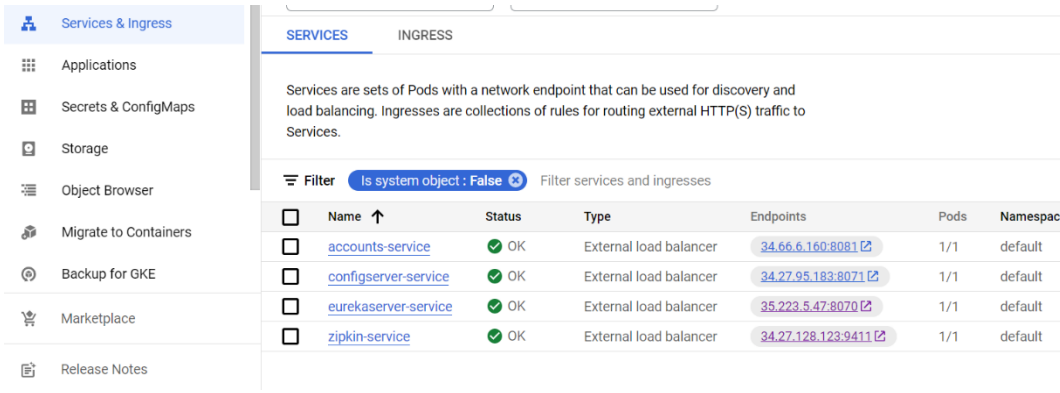
```
:\Users\z0034ydd\MicroService-Training-Latest\accounts\kubernetes>kubectl get servi
AME                  TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
ccounts-service      LoadBalancer   10.8.10.69    34.66.6.160     8081:30891/TCP
onfigserver-service  LoadBalancer   10.8.3.245    34.27.95.183    8071:32633/TCP
urekaserver-service  LoadBalancer   10.8.5.218    35.223.5.47     8070:32622/TCP
ubernetes            ClusterIP      10.8.0.1      <none>          443/TCP
ipkin-service        LoadBalancer   10.8.1.176    34.27.128.123   9411:30224/TCP
```

We can see that our micro services instances are now exposed as services corresponding ip address, service name and port. This is actually done by kube proxy. We can use above information to invoke operations on micro service instance.

External IP is the ip with which we can connect to instance externally.

- Lets revise the over all deployment procedure with K8s architecture:
  - When we want to deploy a new micro service in K8s then we pass that command via kubectl CLI. This command is passed to Kube API Server running in Master. Kube API Server will pass that command to Scheduler. Scheduler will see which node has bandwidth to deploy a new instance. Once it identifies a worker node then it pass that command to kubelet of that worker node. Kubelet will launch a POD where micro service instance will be running. Kube proxy will then create a service for the instance and expose a port to external world.
  - When we issue command like get pods the Kube API server will pass that command to etcd as etcd is a database holding entire cluster information.
- Verifying if microservices are deployed in K8S:

- Click on services and Ingres link in K8S cluster on left panel.
- On the rendered page click on link provided under Endpoints. This will direct us to microservice:



- Self-healing inside K8s cluster:
  - Add one more replica for accounts microservice
  - For this make change in accounts yaml file and provide no of replicas as 2.
    Upload the new file through gcloud CLI.
    Upon uploading new file K8s will be smart enough to identify the delta between previous and current definition of micro service. Accordingly, it will make changes in configuration without any down time.

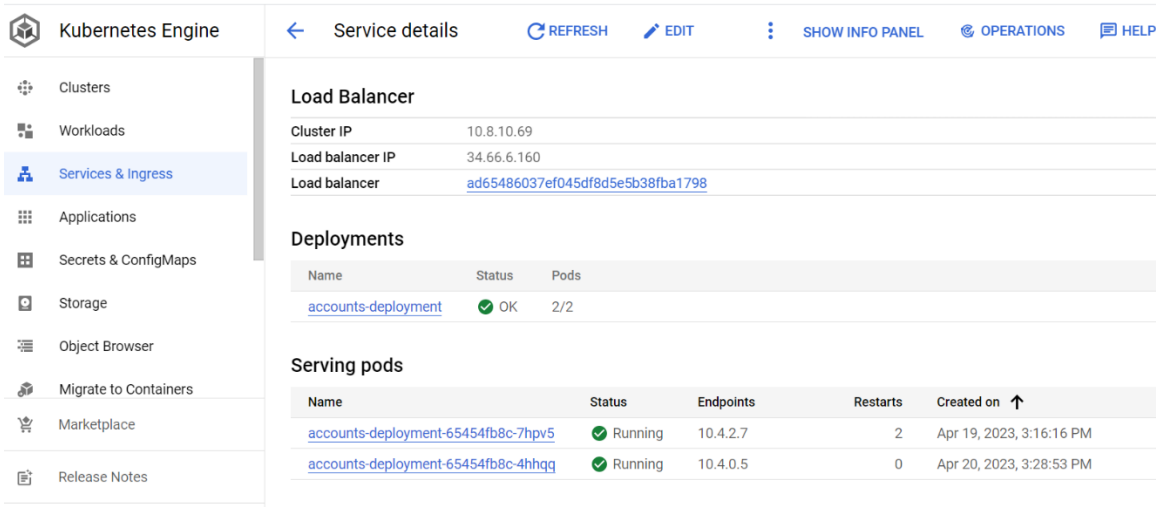
```
C:\Users\z0034ydd\MicroService-Training-Latest\accounts\kubernetes>kubectl apply -f accoun
deployment.apps/accounts-deployment configured
service/accounts-service unchanged
```

    Note that we have changed deployment configuration and we did not change service configration
  - Execute command **kubectl get replicaset**
    Here you will see 2 replicas for accounts microservice
    Also, one more POD will be added for the new replica. But service number will remain same as we have defined service type as LoadBalancer, hence the same Load Balancer will serve via two replicas.

```
C:\Users\z0034ydd\MicroService-Training-Latest\accounts\kubernetes>kubectl get
NAME                                    READY   STATUS    RESTARTS      AGE
accounts-deployment-65454fb8c-7hpv5     1/1     Running   2 (24h ago)   24h
accounts-deployment-65454fb8c-9zwgt     1/1     Running   0             2m2s
configserver-deployment-c664d5577-kfgx7 1/1     Running   0             24h
eurekaserver-deployment-5d7f4f579c-fc59f 1/1    Running   2 (24h ago)   24h
zipkin-deployment-6d588859dc-pwgnj      1/1     Running   0             24h
```

- Delete one of the pods using command **kubectl delete pod <pod-name>.**
  You will see that as soon as we delete one pod, another pod will take its place via command from Controller Manager.

- Click on the accounts service link and scroll down. You will see that accounts service load balancer is serving two PODS:

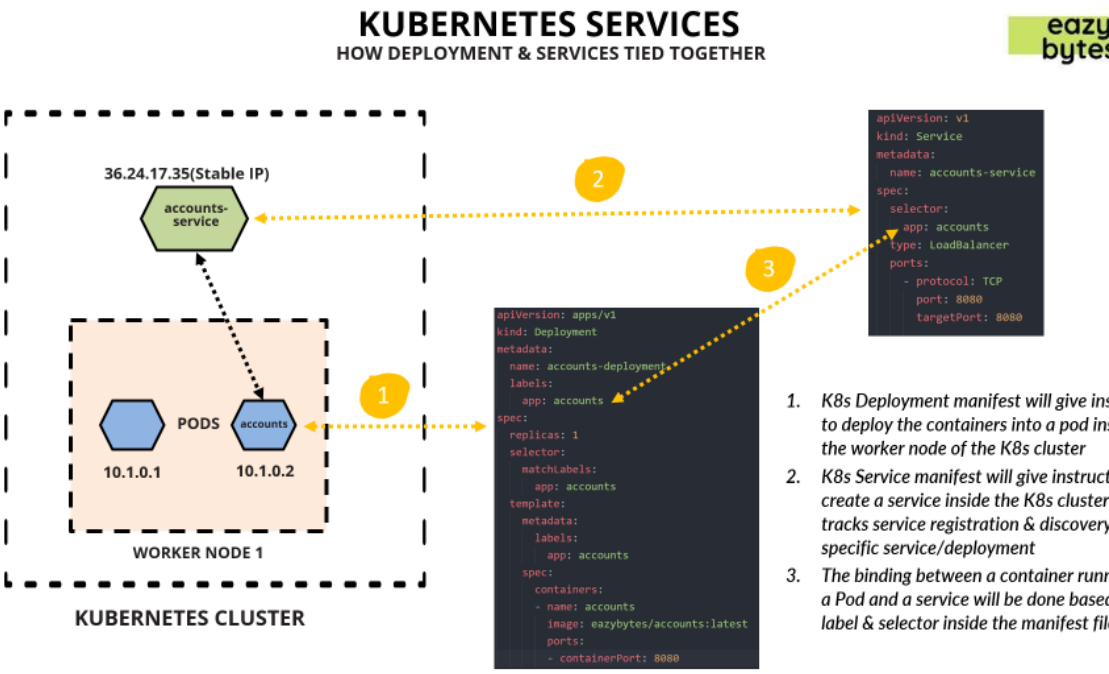


- Logging and Monitoring inside K8s:
  - To view logs in CLI, issue command **kubectl logs <pod-name>**
  - In Cloud console, click on your cluster, scroll down and you will see two links: **ViewLogs, ViewGKEDashboard**
    These links you can use to see individual microservices/pods logs and their status.
    You can use queries to filter logs based on your search criteria.
    - You can check CPU utilization, memory and other metric at node, pod, container level
    - You can also get details for other components like deployment using this feature
    - Organization which do not want to pay heavy cost for K8s can use Open Source frameworks like Grafana, Prometheus for monitoring purpose.
    - Please note that since we are using Zipkin for log aggregation. K8s cluster uses zipkin in background to get log details
    - This Logging and monitoring is available in our cluster as we enabled Log Stack in our configuration of cluster.

- Autoscaling using HPA:
  HPA stands for Horizontal POD Autoscaling. Using this feature K8s automatically scales up or scale down pods based on certain metrics like CPU etc.

  - Execute command **kubectl get hpa** to get current hpa configuration
  - Execute command:
    kubectl autoscale <deployment-name> --min=3 –max=10 --cpu-percent=70

    Above command will autoscale a given deployment to increase number of pods when cpu percentage is greater than 70. It will also make sure that minimum no of pods are 3 and maximum is 10.
- Link between deployment and service:
  - Initially Worker Node is empty
  - We upload yaml file with deployment information
  - Using that information image is downloaded and container is launched in pod
  - PODS get scaled up or scaled down based on configuration
  - Container ip address will keep on changing in case when PODS are added or the existing ones are relaunched due to some issue.
  - External clients have no clue about these dynamic ip addresses and thet are internal to K8s.
  - This is where service comes in to play.
  - Using service information in yaml file, K8s creates a service which acts as a registry
  - In this registry containe ip addresses, port information is registered
  - Service type is Load Balancer which has a static ip address.
  - Clients invoke request upon Load Balancer ip address which automatically routes request to container keeping traffic balanced.
  - See snap shot below:



KUBERNETES: Deploying all Micro services in K8s:

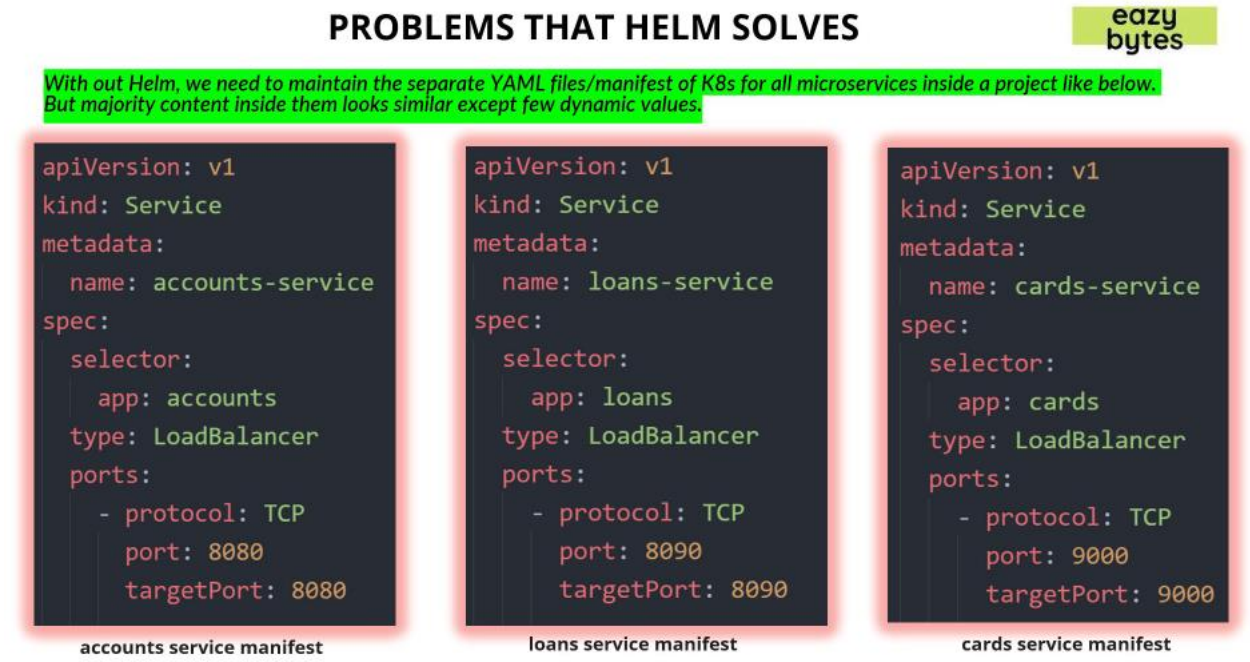
A) Create yaml files for remaining micro services i.e. cards and loans
B) Upload them in K8s using Google cloud CLI
C) Validate them if they are all up and running

- Problem with manual creation of yaml or manifest files:
  - For multiple micro services, we need to create multiple yml files which is painful
  - For multiple yml files we need to upload them in K8s individually which is also painful (through apply command)
    This is where Helm charts come into play.

HELM CHARTS

A) Introduction:
   - Helm is a package manager for K8s
   - Package manager is a component used to install/update software modules
   - For example, pip is a package manager for Python, npm is package manager for Node
   - Similarly, Helm is a package manager fot K8s which facilitates deployment of multiple microservices in K8s.
   - Helm chart will contain definitions of all micro services deployment in a single file.

- If we see yml files of our services, there will be common content which is applicable in each file and then there is some dynamic content. Helm takes advantage of this common static content and facilitates creation of a common template yml file which is applicable for all micro services:



See above example where we can observe that there is a common/static content among all yml files

- Structure of common template file and dynamic values file:

PROBLEMS THAT HELM SOLVES

eazy bytes

With Helm, we can a single template yaml file like shown below. Only the dynamic values will be injected during K8s services setup based on the values mentioned inside the values.yaml present inside each service/chart.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.deploymentLabel }}
spec:
  selector:
    app: {{ .Values.deploymentLabel }}
  type: {{ .Values.service.type }}
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
```

Helm service template

```yaml
deploymentLabel: accounts
service:
  type: ClusterIP
  port: 8080
  targetPort: 8080
```

values.yaml

We create values, yaml file for each micro service which gets injected into common template file to create final yaml file for the service. All this is done internally by Helms framework.

- Problem that Helm solves:

PROBLEMS THAT HELM SOLVES

eazy bytes

HELM SUPPORTS PACKAGING OF YAML FILES

With the help of Helm, we can package all of the YAML manifest files belongs in to an application into a Chart. The same can be distributed into public or private repositories.

HELM SUPPORTS EASIER INSTALLATION

With the help of Helm, we can set up/upgrade/rollback/remove entire microservices applications into K8s cluster with just 1 command. No need to manually run kubectl apply command for each manifest file.

HELM SUPPORTS RELEASE/VERSION MANAGEMENT

Helm automatically maintains the version history of the installed manifests.

Due to that rollback of entire K8s cluster to the previous working state is just a single command away.

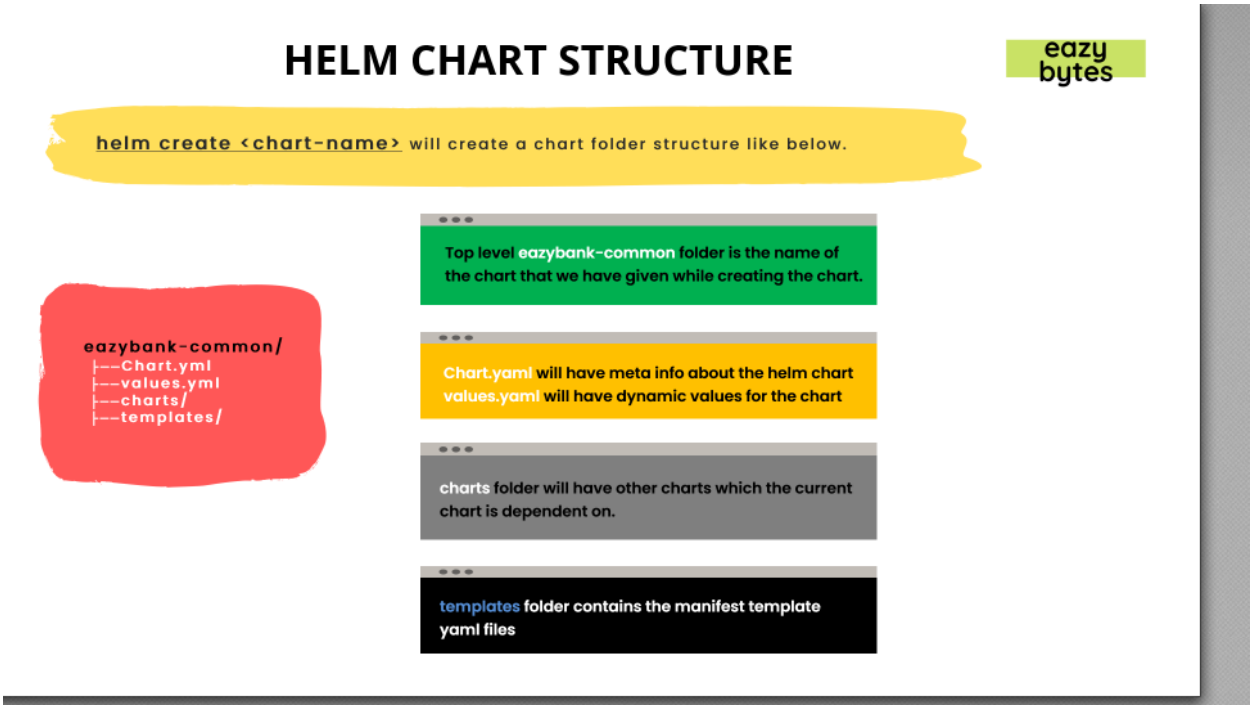
HELM

Helm acts as a package manager for K8s. Just like a package manager is a collection of software tools that automates the process of installing, upgrading, version management, and removing computer programs for a computer in a consistent manner, similarly Helm automates the installation, rollback, upgrade of the multiple K8s manifests with a single command.

B) Installing Helm:

- Navigate to website: https://helm.sh/
- Click on Get started and then click on Installation.
- We need to have chocolatey windows package manager to install helm:
  - Navigate to website https://chocolatey.org/
  - Click on Install option
  - Choose Individual installation
  - Follow the steps as mentioned in website and install chocolatey package manager
- Now execute command:
  choco install kubernetes-helm

C) Creating First helm Chart:
- Execute command **helm version**
- Execute command helm create <chart-name>
- Once you execute this command then a folder named <chart-name> will be created. Inside that folder there will be come default folder and files created by Helm framework.



D) Installing the default Helm chart into K8s cluster:
- Connect to K8s cluster
- Execute command **helm ls,** to get details of current heml charts deployed in K8s cluster (How helm is able to connect to K8s? When we connect to K8s cluster then a local file is created <User-folder>/kube/config). This file contains details of K8s cluster connection. Helm uses very same file to talk to K8s cluster
- Navigate to folder where helm files are present
- Execute command **helm install <deployment-name> <helm folder name>**

D) Create Helm chart for our Example Micro services

In this section we will learn how to create Helm chart for our business micro services:

- Please note that this helm files are written in Go language
- Create service.yaml template file:

```
{{- define "common.service" -}}
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.deploymentLabel }}
spec:
  selector:
    app: {{ .Values.deploymentLabel }}
  type: {{ .Values.service.type }}
  ports:
    - name: http
      protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
{{- end -}}
```

- Create deployment.yaml template file:

```
{{- define "common.deployment" -}}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.deploymentName }}
  labels:
    app: {{ .Values.deploymentLabel }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Values.deploymentLabel }}
  template:
    metadata:
      labels:
        app: {{ .Values.deploymentLabel }}
    spec:
      containers:
      - name: {{ .Values.deploymentLabel }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

```
        ports:
        - containerPort: {{ .Values.containerPort }}
          protocol: TCP
        env:
        {{- if .Values.profile_enabled }}
        - name: SPRING_PROFILES_ACTIVE
          valueFrom:
            configMapKeyRef:
              name: {{ .Values.global.configMapName }}
              key: SPRING_PROFILES_ACTIVE
        {{- end }}
        {{- if .Values.zipkin_enabled }}
        - name: MANAGEMENT_ZIPKIN_TRACING_ENDPOINT
          valueFrom:
            configMapKeyRef:
              name: {{ .Values.global.configMapName }}
              key: MANAGEMENT_ZIPKIN_TRACING_ENDPOINT
        {{- end }}
        {{- if .Values.config_enabled }}
        - name: SPRING_CONFIG_IMPORT
          valueFrom:
            configMapKeyRef:
              name: {{ .Values.global.configMapName }}
              key: SPRING_CONFIG_IMPORT
        {{- end }}
        {{- if .Values.eureka_enabled }}
        - name: EUREKA_CLIENT_SERVICEURL_DEFAULTZONE
          valueFrom:
            configMapKeyRef:
              name: {{ .Values.global.configMapName }}
              key: EUREKA_CLIENT_SERVICEURL_DEFAULTZONE
        {{- end }}
    {{- end -}}
```

- Create configmaps.yaml file:

```
    {{- define "common.configmap" -}}
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: {{ .Values.global.configMapName }}
    data:
      MANAGEMENT_ZIPKIN_TRACING_ENDPOINT: {{ .Values.global.zipkinBaseURL }}
      SPRING_PROFILES_ACTIVE: {{ .Values.global.activeProfile }}
      SPRING_CONFIG_IMPORT: {{ .Values.global.configServerURL }}
```

EUREKA_CLIENT_SERVICEURL_DEFAULTZONE: {{ .Values.global.eurekaServerURL }}
{{- end -}}

- Creating Accounts microservice helm chart:
  - Create a folder where you want to create helm charts for business micro services
  - Navigate to above folder and execute command **helm create accounts (Please note that for creating common template files we have to follow the same procedure before actual creation of common template files. The above command create default helm charts file but they are specific to nginx server. Delete all files under template folder and empty content of values.yaml, then start creating actual helm files.)**
  - Above command will create a folder named account. In account folder there will charts.yaml, values.yaml files

    Under account folder there will be templates folder.

  - Open the charts.yaml file. There will be some default entries, do not change it. Add given below entry to import common template chart files:
    dependencies:
    - name: eazybank-common
    version: 0.1.0
    repository: file://../../eazybank-common
  - Under templates folder we will create template for deployment and service for accounts microservice
  - Under template folder, create file deployment.yaml. Provide given below content:
    {{- template "common.deployment" . -}}
    Above instruction will import common.deployment template

  - Under template folder, create file service.yaml. Provide given below content:
    {{- template "common.service" . -}}
    Above instruction will import common.service template
  - Now we have defined common templates and imported them as well. Lets now create values.yaml.
    Under accounts folder, provide given below entries:
    # This is a YAML-formatted file.
    # Declare variables to be passed into your templates.

    deploymentName: accounts-deployment
    deploymentLabel: accounts

    replicaCount: 1

    image:

```
  repository: eazybytes/accounts
  tag: latest

containerPort: 8080

service:
  type: LoadBalancer
  port: 8080
  targetPort: 8080

config_enabled: true
zipkin_enabled: true
profile_enabled: true
eureka_enabled: true
```

As you can see we have defined all dynamic values for our accounts chart

- To validate if all dependencies and charts are created correctly, navigate to accounts folder and execute command **helm dependency build**

  On successful validation, under accounts/chart folder a .gz file will be created which will contain accounts microservice final chart

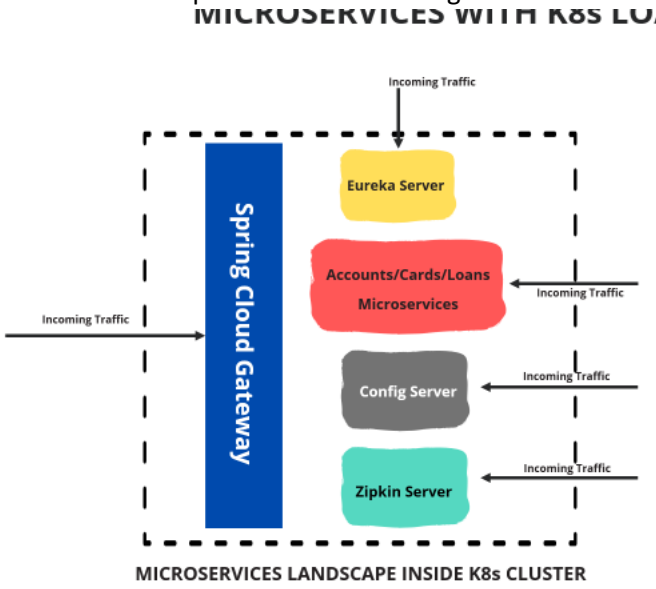
- Follow above steps to create helm charts for other micro services.

E) Uploading Helm Chart in K8s cluster:
We have now created helm charts for individual micro services. Its time to upload them in K8s. But again, do we still need to upload individual charts in K8s cluster, No.

- We will create another chart one for each environment like Prod, Dev etc.
- These environmental charts will contain entries for all helm charts.
- Thus, by uploading a single chart we can deploy all micro services in K8S.

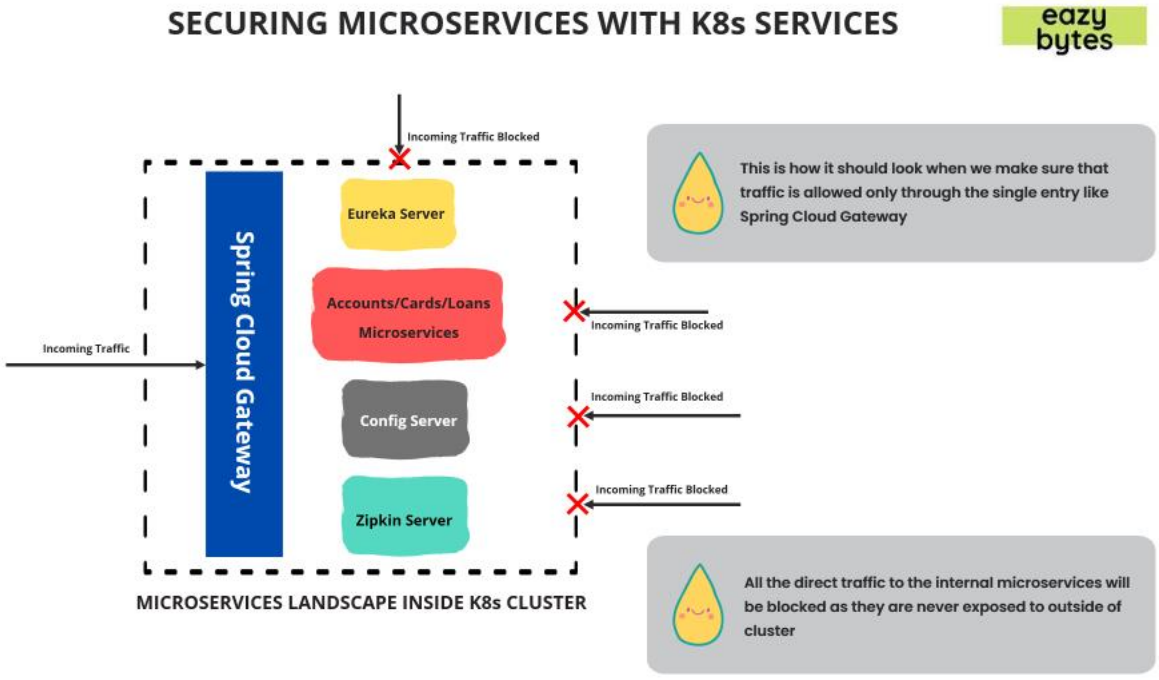
SECURING MICROSERVICE USING K8S SERVICE

A) With Load Balancer as service type, all our micro services in K8s cluster are exposed to outside world. The set up looks like below diagram:



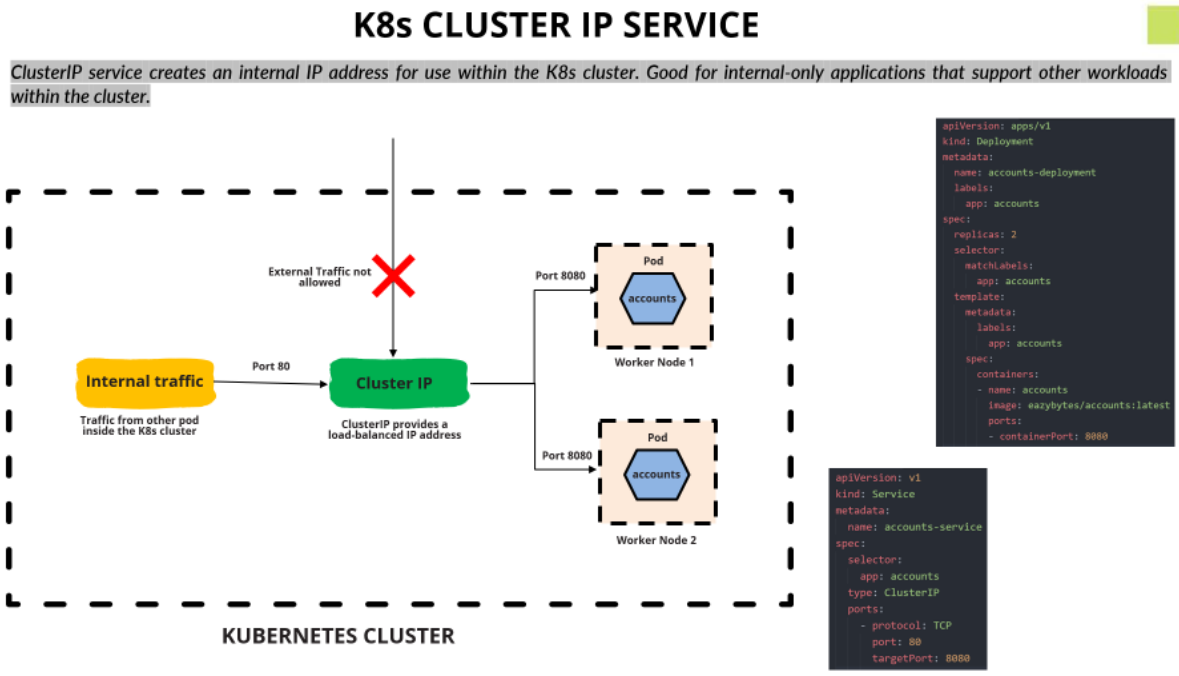
Even with presence of Gateway server, any one can access micro service as Load Balancer exposes a public API

B) The required set up should be like below:



C) Cluster IP Service:

- Cluster IP Service exposes an internal IP Address for micro service
- This can be used for Use cases where micro services needs to communicate internally. For example, accounts micro service invoking cards micro service to get card details for customer.
- Cluster IP Service provides a Load Balanced IP Address. See diagram below:



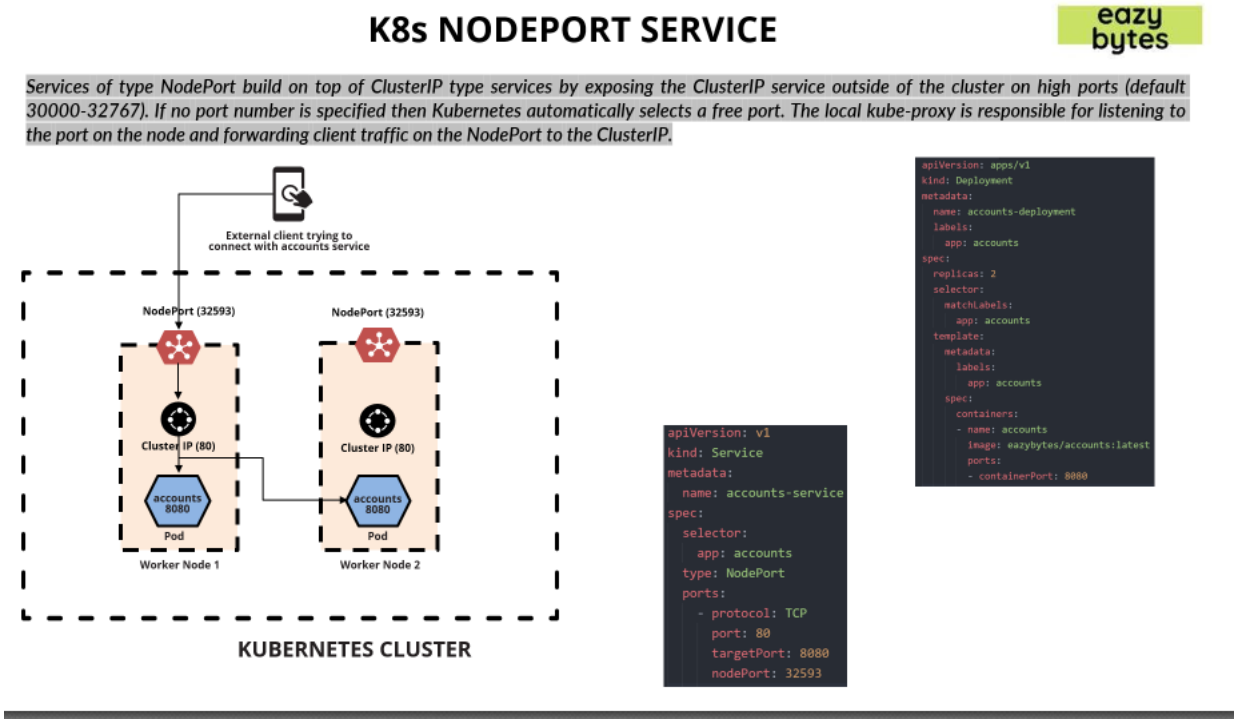
- By default, if we do not mention service type then it is taken as Cluster IP
- Steps to use Cluster IP in our Use case:
  - Change Service type of config server, eureka server, accounts, loans, zipkin to Cluster IP. DO NOT Change service type of gateway server as we need to access Gateway Service externally.

D) Node Port Service:

- Micro service are accessed via <workerNode IP Address>:<port number>.
- For accessing micro service you need to know worker node ip address
- Port number can be explicitly defined in yaml/helm chart file, otherwise a default port number is assigned by K8s cluster
- By default GCP will not allow traffic on this port. TO enable the traffic, you need to execute below command:
  gcloud compute firewall-rules create <node-port name> --allow tcp:<port number>
- Internally the requested is forwarded via Cluster IP Load Balancer, hence if you access micro service via worker node 1 ip address then its not necessary that request will

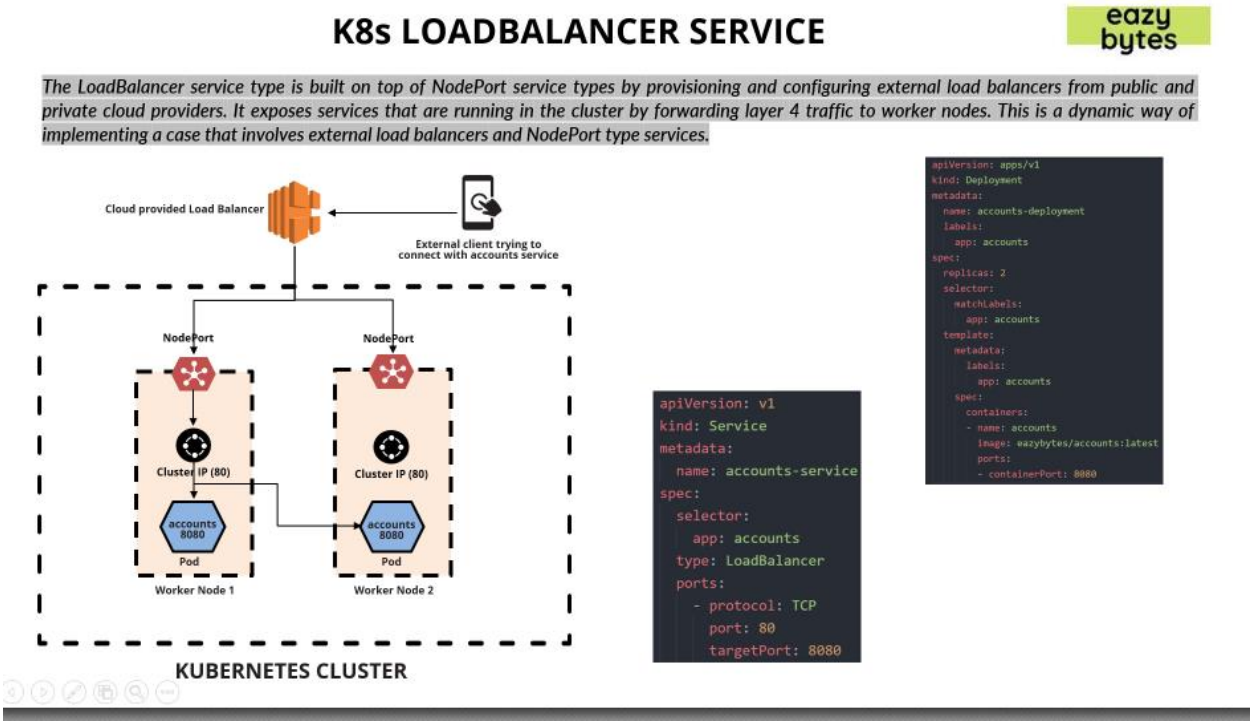
always go to worker node1, it will depend on what worker node cluster ip load balancer has chosen

- You need to run several commands in kubectl to get worker node and the port where micro service is deployed
- Node port service is rarely used, either Load Balancer or Cluster IP is used.



E) Load Balancer Service:

- Load Balancer Service internally uses Node Port Service and Cluster IP Service.
- Hence the most basic service is Cluster IP Service. On top of Cluster IP, Node Port Service is built. On top of Node Port Service, Load Balancer Service is built



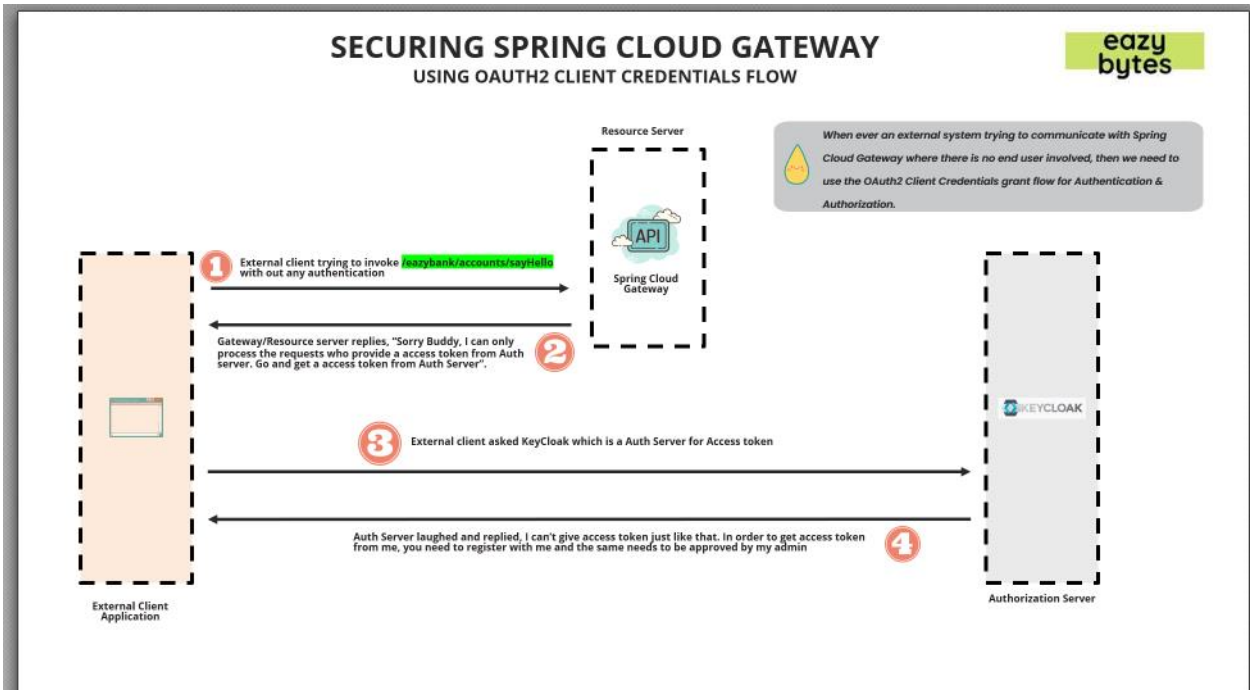
SECURING USING OUTH2 FRAMEWORK

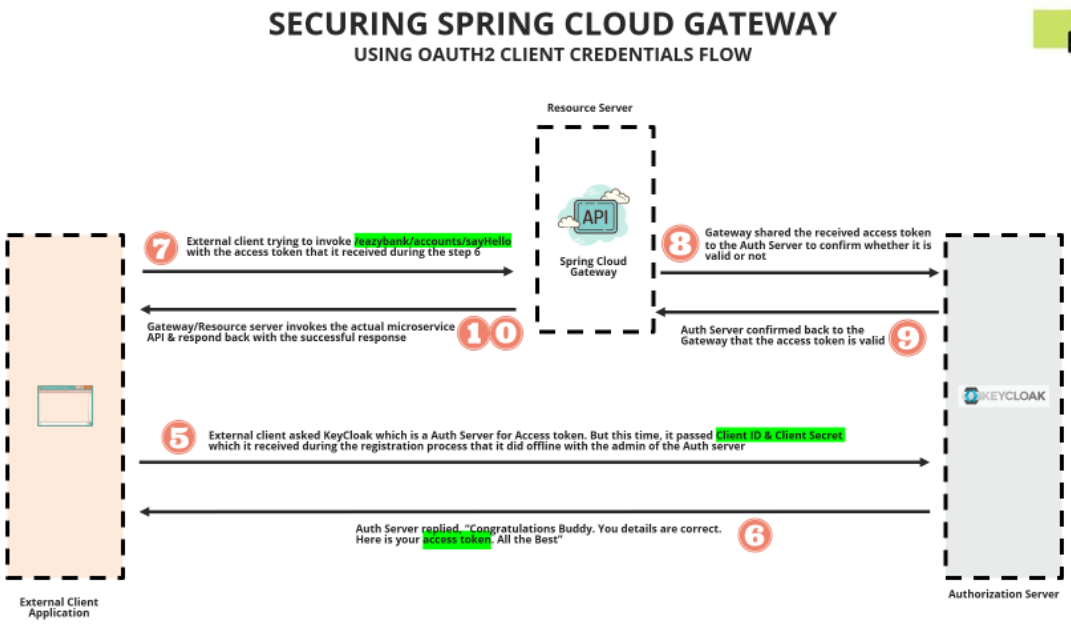
A) We are still able to secure our Micro service using Spring Cloud GW and Cluster IP Address but still we do not know if requested user is an authenticated user.
B) We can use OAuth 2 Framework for the same. We can use KeyCloak Framework
C) Introduction to OAuth 2 Framework:
   - Basic Authentication and drawbacks:
     - In basic authentication you provide a user name and password in an html form
     - The request goes to back end server where credentials are validated from DB.
     - If user is valid then a session is generated, as long as session is valid, the user will be allowed to access.
     - For session cookies are used
     - Its not a good solution when you have rest based interfaces or in micro service ecosystem
     - Its also not a good solution when a third party needs to access your API

   - OAuth 2 Framework:
     - OAuth2 keeps authentication logic in a separate authentication server. Hence authentication is coupled with business logic
     - It has 4 to 5 grant flows.

- Authorization Grant Flow: Used when end user is involved. For example a user trying to access micro service
- Client Credential Grant Flow: Used when one application wants to access other. For example internal micro service communication.
- Once we send user credentials or client credentials to auth server then it validates it and issues a token. Based on token one can access all applications in a secured manner
- SSO with OAuth2: OAuth2 supports SSO. In an enterprise or an Organization there can be multiple apps, mobile apps, web apps. If all of them uses or point to same auth server then wit ha single token we can jump from one app to other without authenticating every time. This is known as SSO. OAuth Framework provided this.
- OAuth also support authentication from third party applications like FB or Gmail. For example lets say you long into a web app. Now instead of you providing your name, surname, address, email id the web app exposes a social login like gmail or FB. So your web app will fetch your name, surname etc from social app details. In this case your are not providing your social app user name and password to third party web app. Once you login into gmail then gmail will issue a token, using that token third party web app will fetch name, surname, address and other details.

B) OAuth 2 Flow:

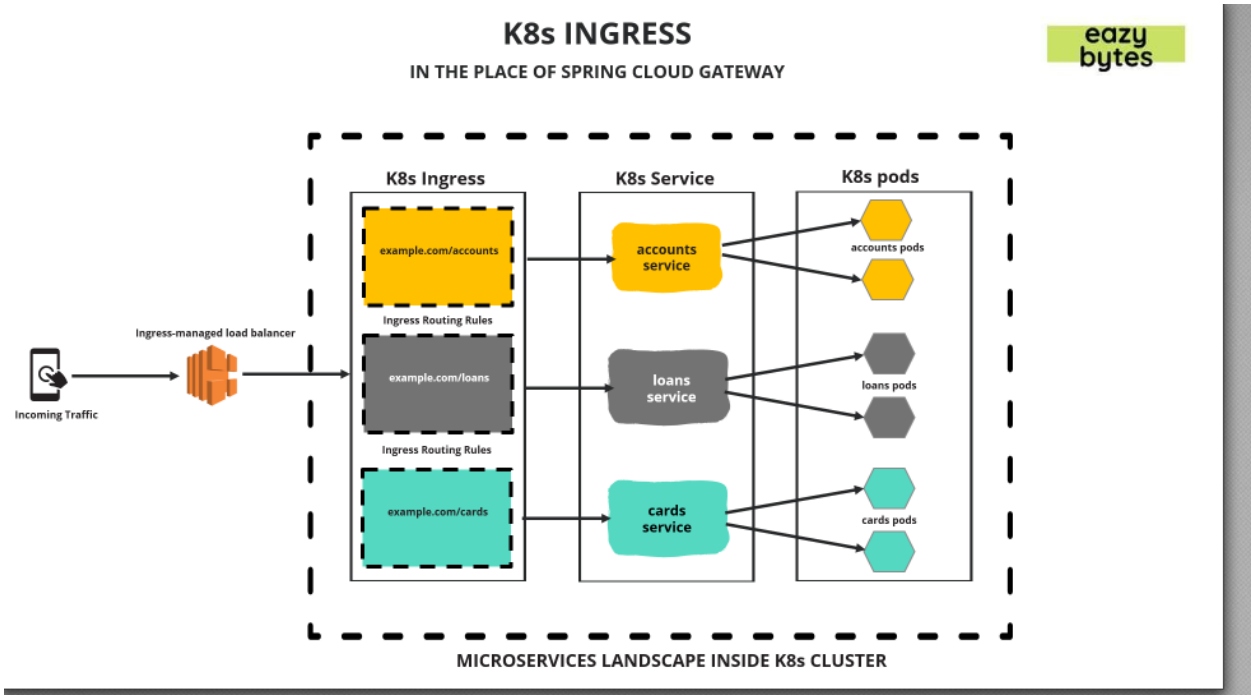




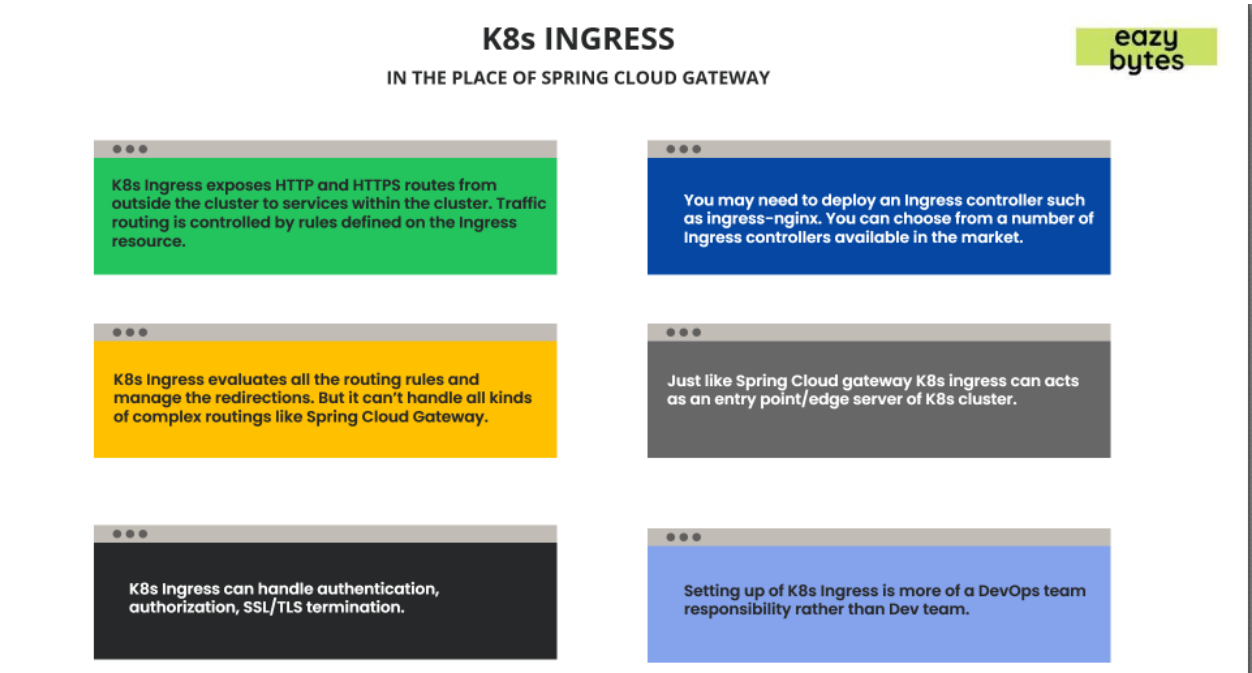
K8S Ingress & Service Mesh

A) Ingress:

- An alternative to Spring Cloud Gateway in K8S deployment
- With Ingress you won't require Spring Cloud Gateway
- It will act as an edge server



-



-
-

B) Service Mesh:

- With Service Mesh you won't require Sleuth, Zipkin, OAuth 2 implementation to monitor and secure micro services