Monolithic & SOA & MICRO-SERVICE:

1. Monolithic:

Adv :

1. It is best for small project or app
2. fewer cross-cutting concerns like logging, auditing
3. better performance due to no network latency.

DisAdv :

1. Not able to adopt new technologies
2. single code on single server with one database
3. tiny update need full deployment
4. any change will affect other layers.
5. SOA – Service oriented Architecture

Adv :

1. we can make different services on different server

2. better maintainability and reliability and parallel development.

DisAdv:

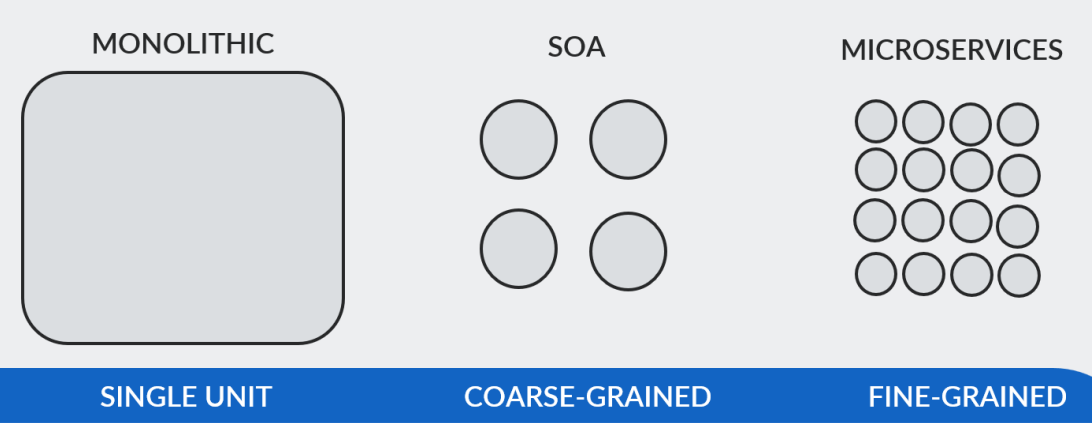
1. Complex management due to maintain proxy for connecting all services and SOAP webservices.
2. high investment costs because using enterprise oracle service bus and extra overload.
3. Microservices:

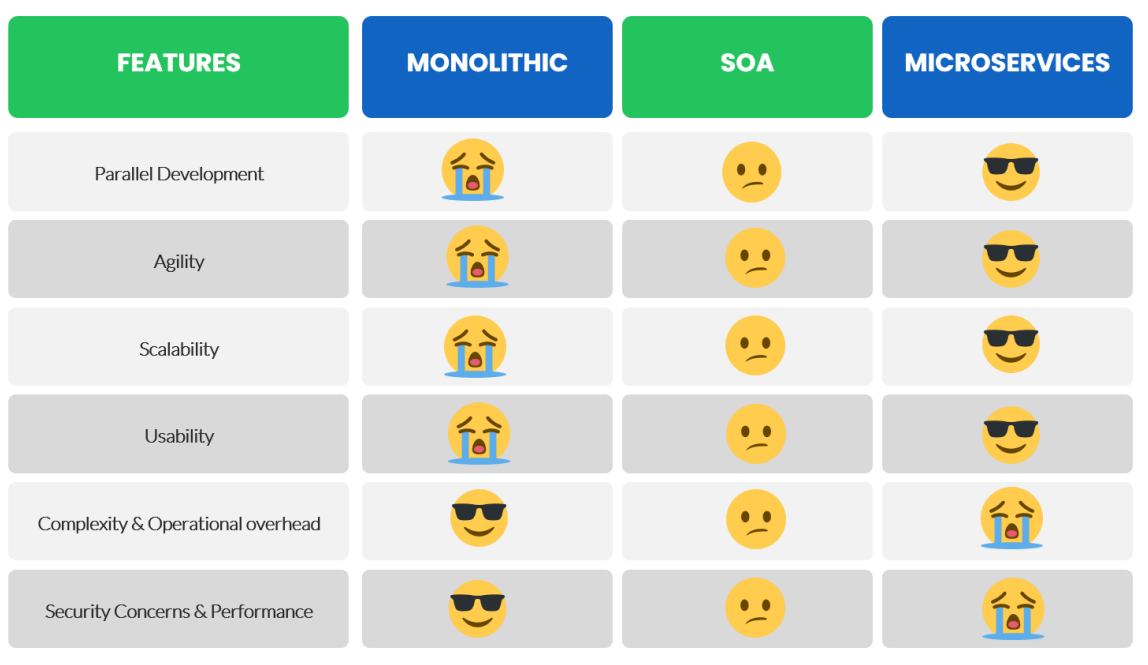
Adv :

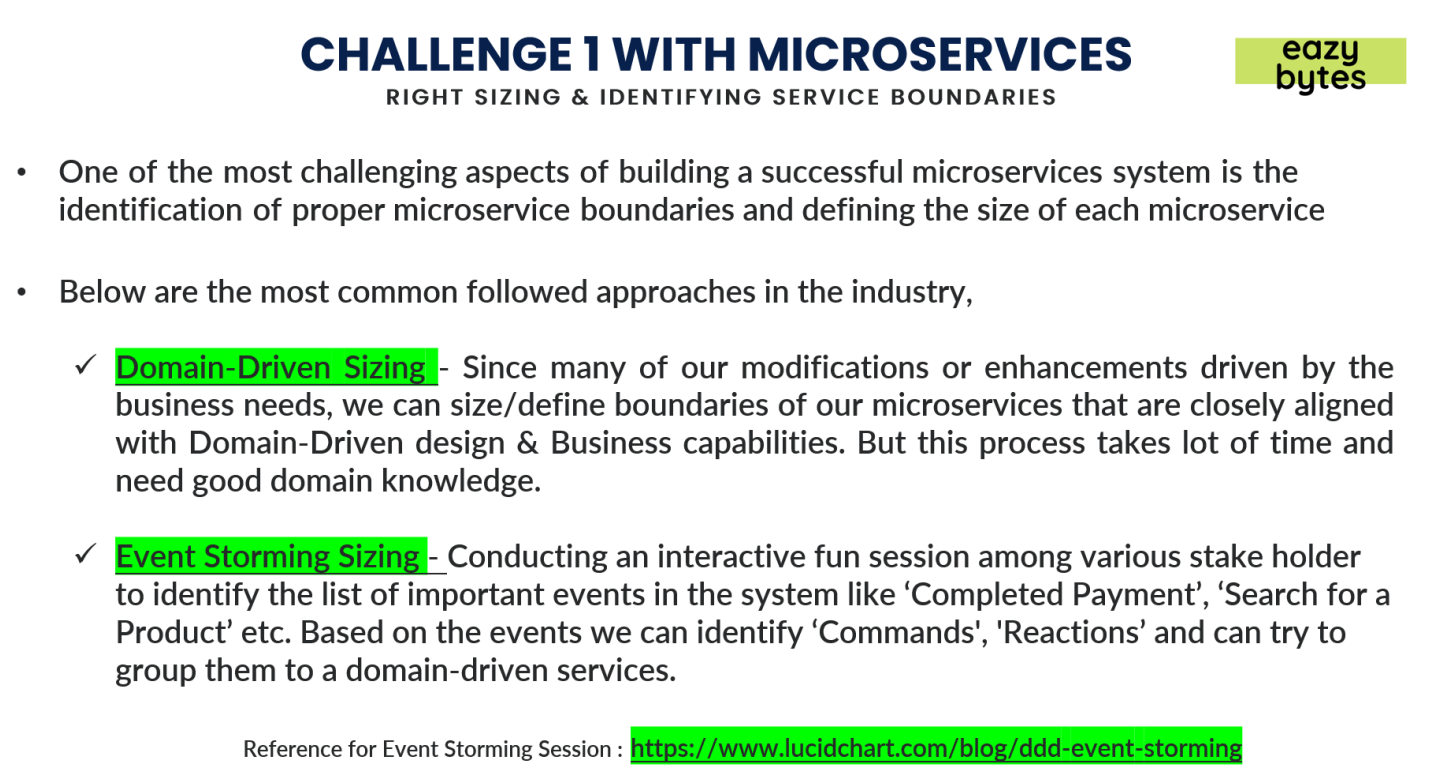
1. Easy to develop, test and deploy
2. Parallel development
3. Ability to scale horizontally like you want to scale one of your services you can do easily

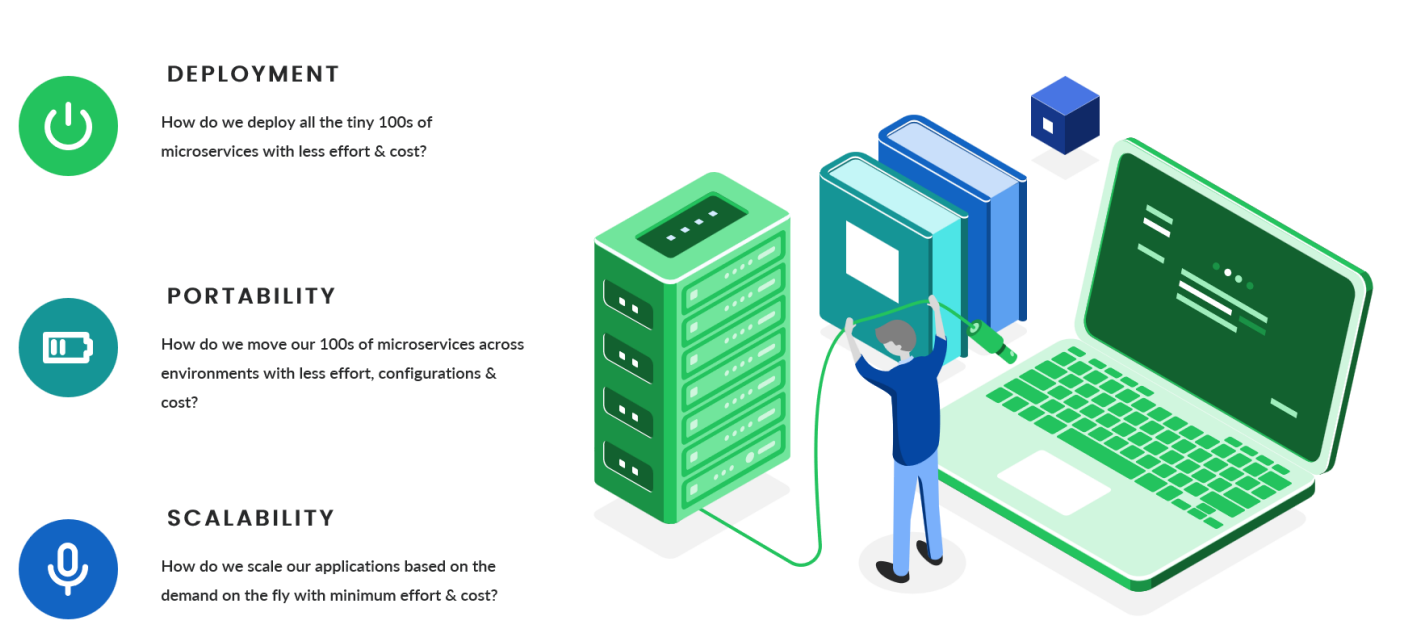
DisAdv :

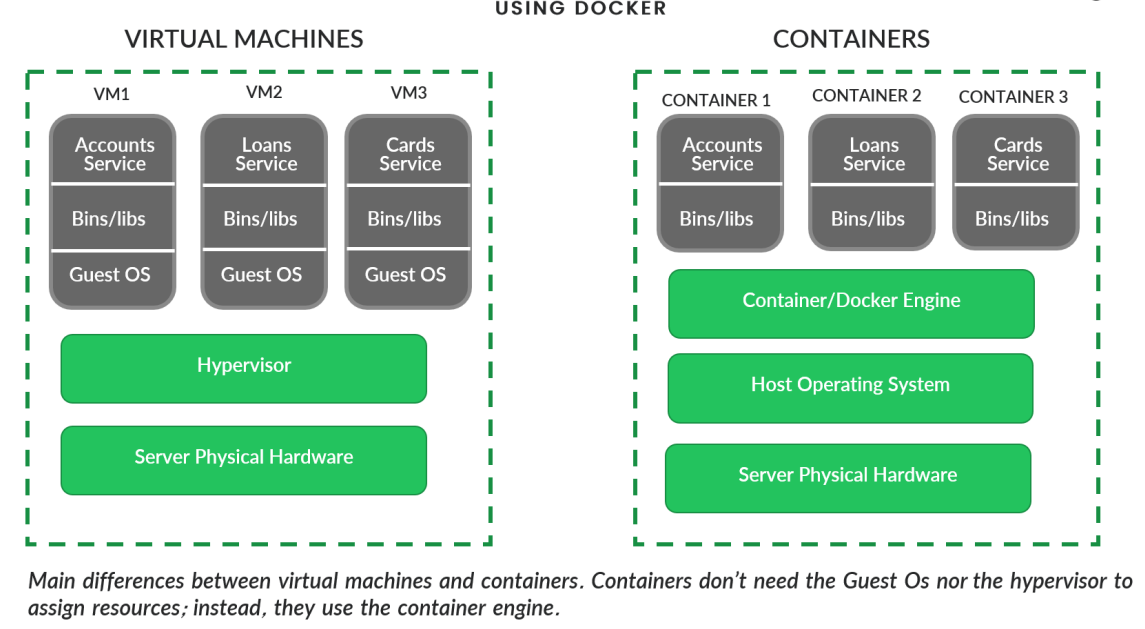
1. Complexity – manage all instances of servers but this can be solved by azure, aws, docker etc.
2. Infrastructure overhead and security concerns.

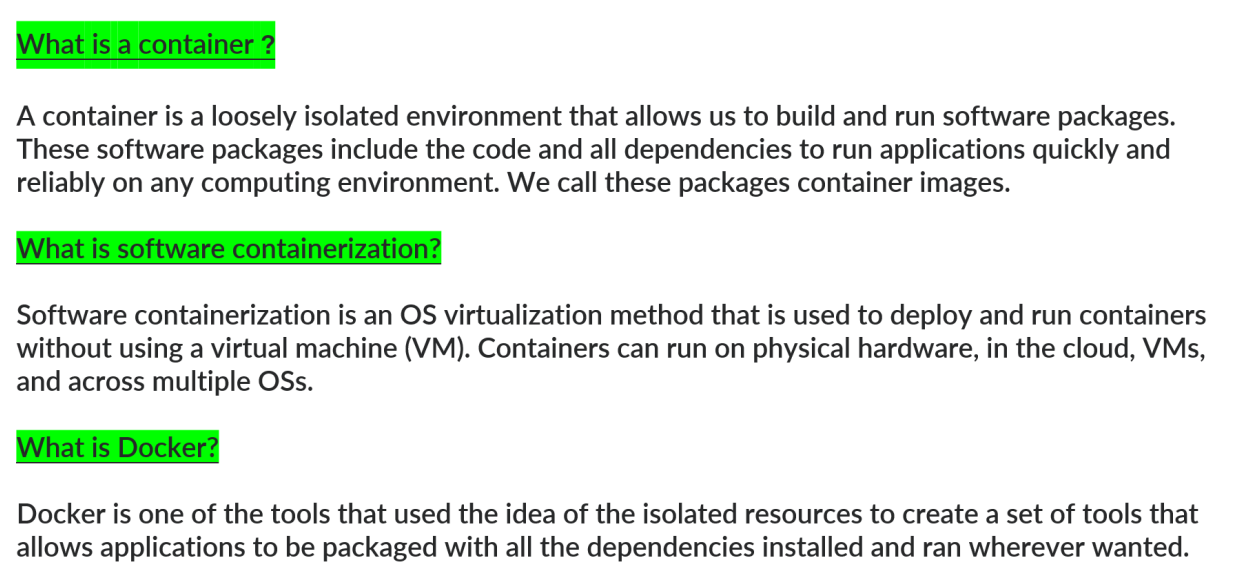


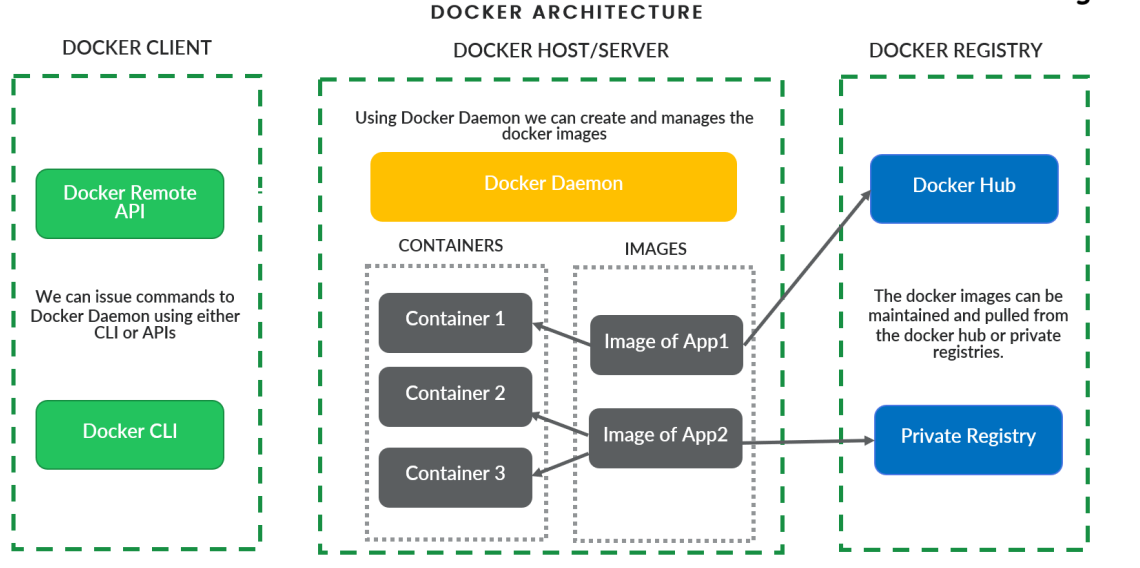












To run spring boot project on cmd

* mvn clean install
* mvn spring-boot:run

if not mvn use java = java -jar target/Accounts-0.0.1-SNAPSHOT.jar

1. Accounts :

* Create Dockerfile ouside src folder
* Create images : docker images
* docker build . -t ingleajay/accounts
* docker image inspect imageid
* docker run -p 8081:8080 ingleajay/accounts
* docker ps = for containers view
* docker stop containerid = stop container
* docker logs containerid = app logs
* docker start containerid1 containerid2
* docker container pause containerid
* docker container unpause containerid
* docker container inspect containerid
* docker stats
* docker rm container id
* docker image imageid

We can create images with any no of container with any port

It means we are able to deploy application on different instances .

What is BuildPacks : Transform source code into images

Without need of dockerfile in application we can create images

In java we use paketo buildpacks for doing this stuff

In loans :

Without creating docker file : first do – mvn clean install

mvn spring-boot:build-image -Dspring-boot.build-image.imageName=ingleajay/loans

Add images into docker hub

docker push docker.io/ingleajay/loans

Docker Compose :

When we have multiple Microservices it is difficult to create images like this so to manage we have

Docker compose

Your docker compose file in outside of all folder initially – file name: docker-compose.yml

Inside content is :

version: "3.8"

services:

accounts:

image: ingleajay/account:latest

mem\_limit: 700m

ports:

- "8080:8080"

networks:

- ingleajay-network

loans:

image: ingleajay/loans:latest

mem\_limit: 700m

ports:

- "8090:8090"

networks:

- ingleajay-network

cards:

image: ingleajay/cards:latest

mem\_limit: 700m

ports:

- "9000:9000"

networks:

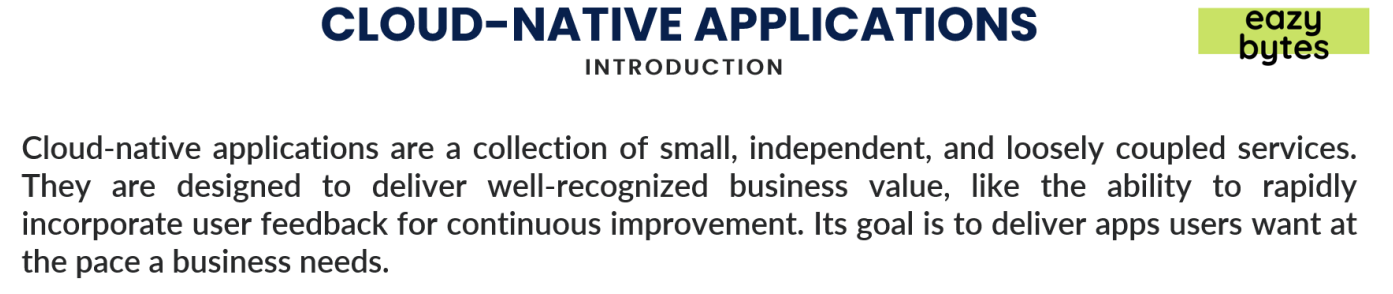
- ingleajay-network

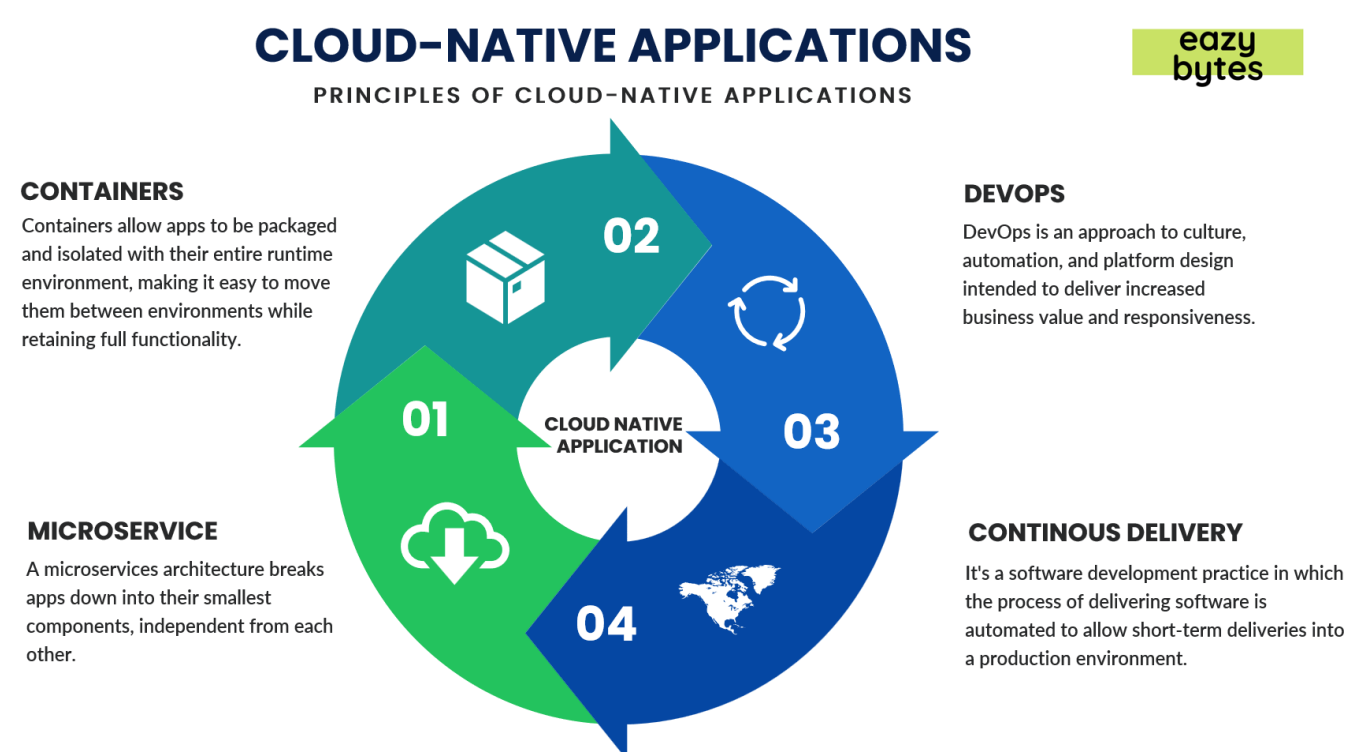
networks:

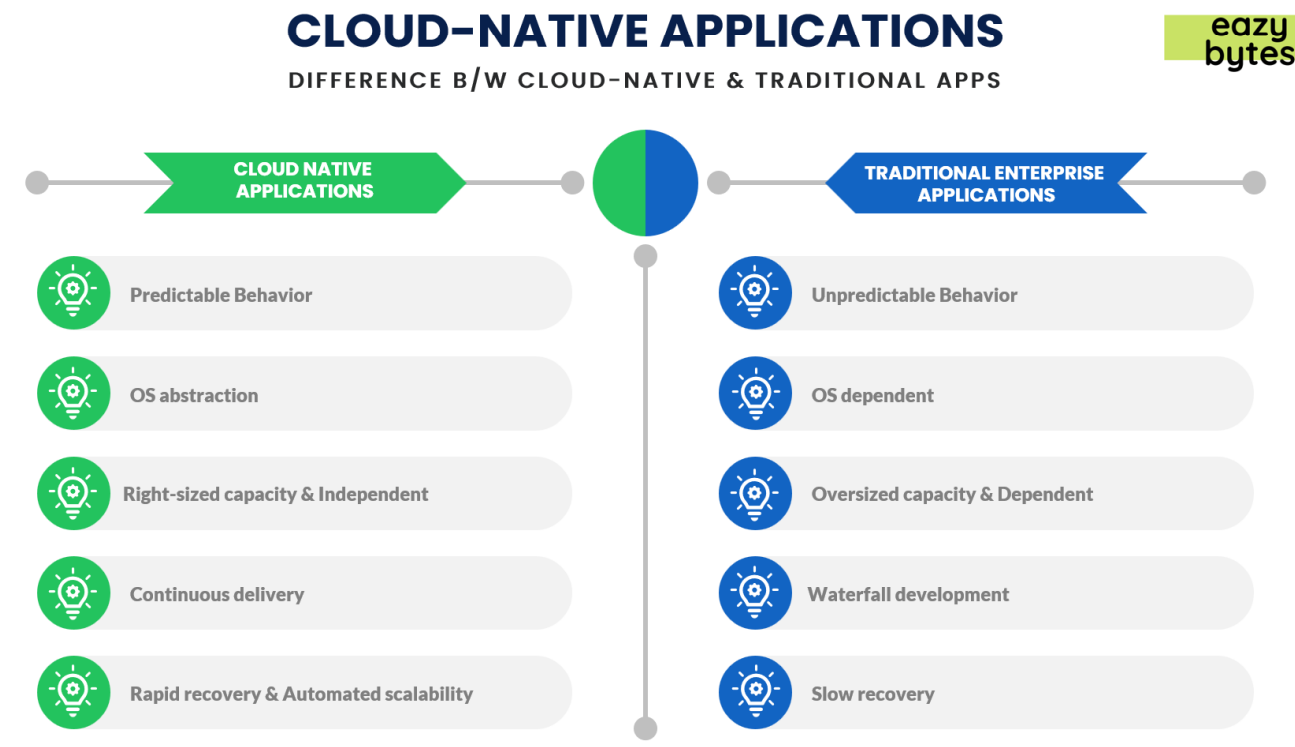
ingleajay-network:

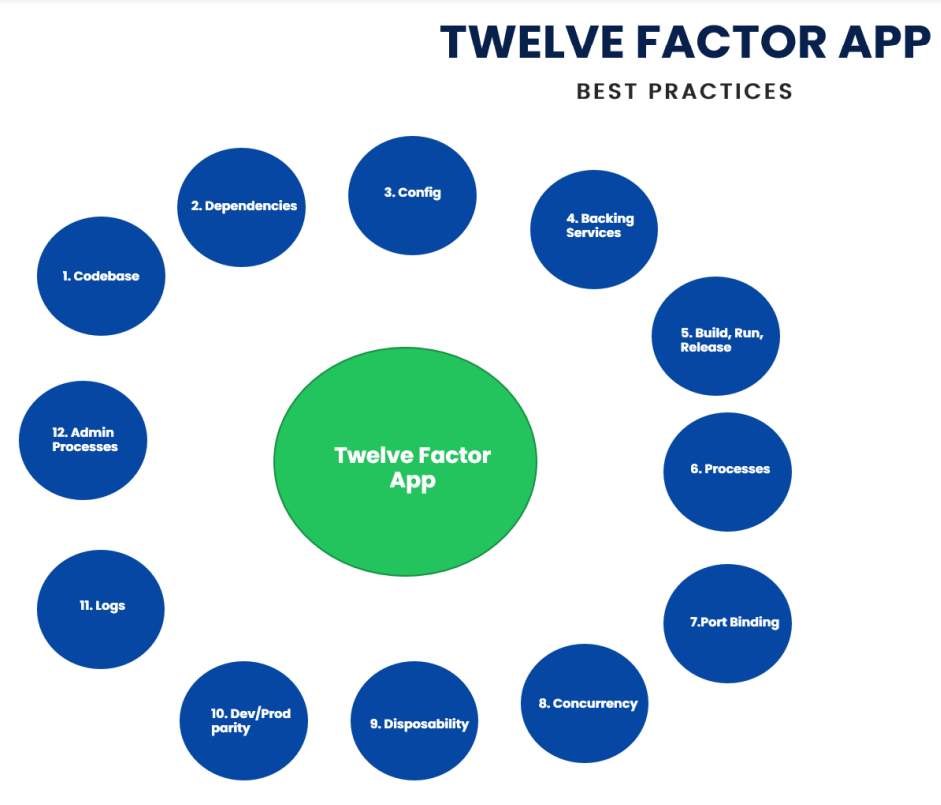
docker-compose up = with this commands all services will run

docker-compose stop



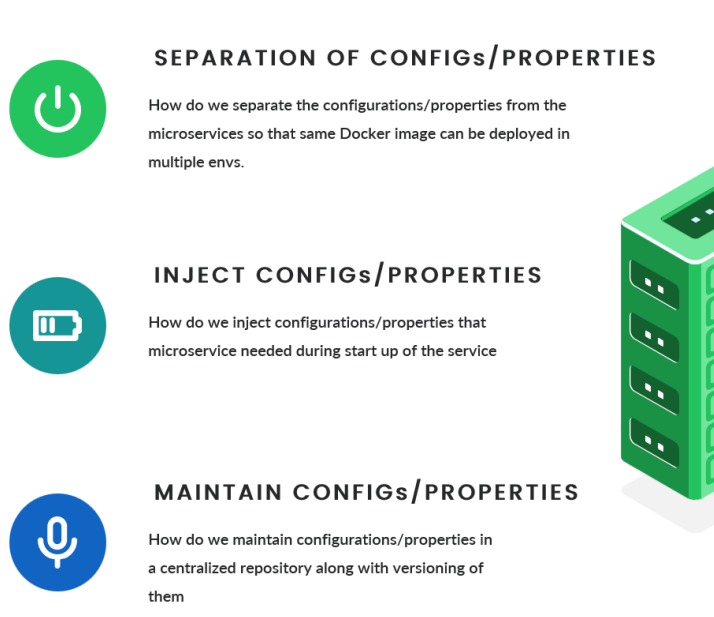


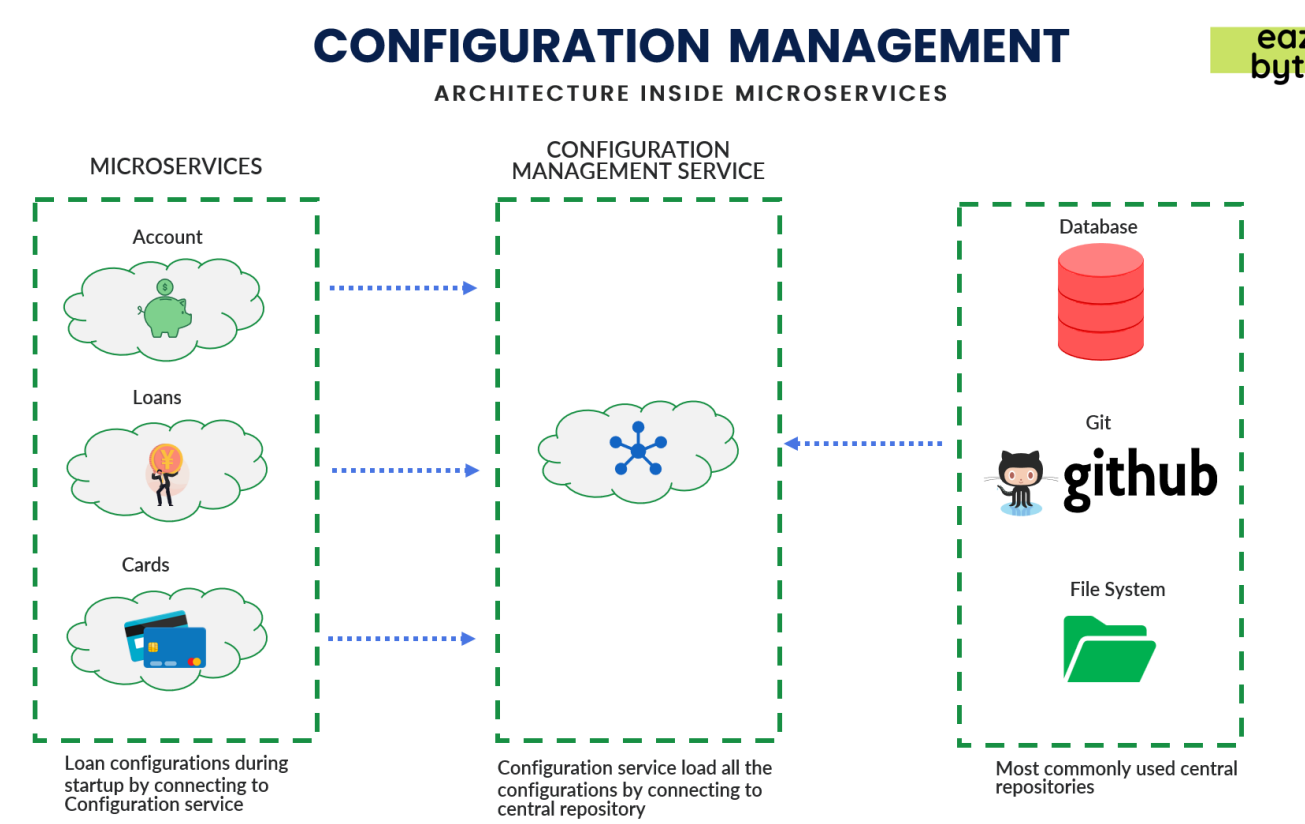




1. codebase – each microservice has it’s own code base like github like accounts and cards are github different
2. dependencies – each microservices has pom.xml and maven or gradle
3. config – it’s store env specific configurations from your code
4. Backing service – change local connections to third party change lke db from local to aws db
5. Build,release, run – it will be like after docker is build it should go in next stage that release & that is different than build stage
6. Processes – if microservice has chaining call for end to end results then it should be persist and has stateful services
7. Port binding – all different micoservices has their own ports
8. Concurrency – when we want to scale up and scale down size of ram then we always follow horizontal approach like it crease instances of app but vertical approach will increase size of hardware
9. Disposability – if microservice is failing to run then that microservice will be disposable and new instance is created
10. Dev/prod parity – we should not change anything in this env manually because it will create problems later.
11. Logs – maintaine logs of each app
12. Admin process – clean up and migrations task

Configuration and properties management in microservices:

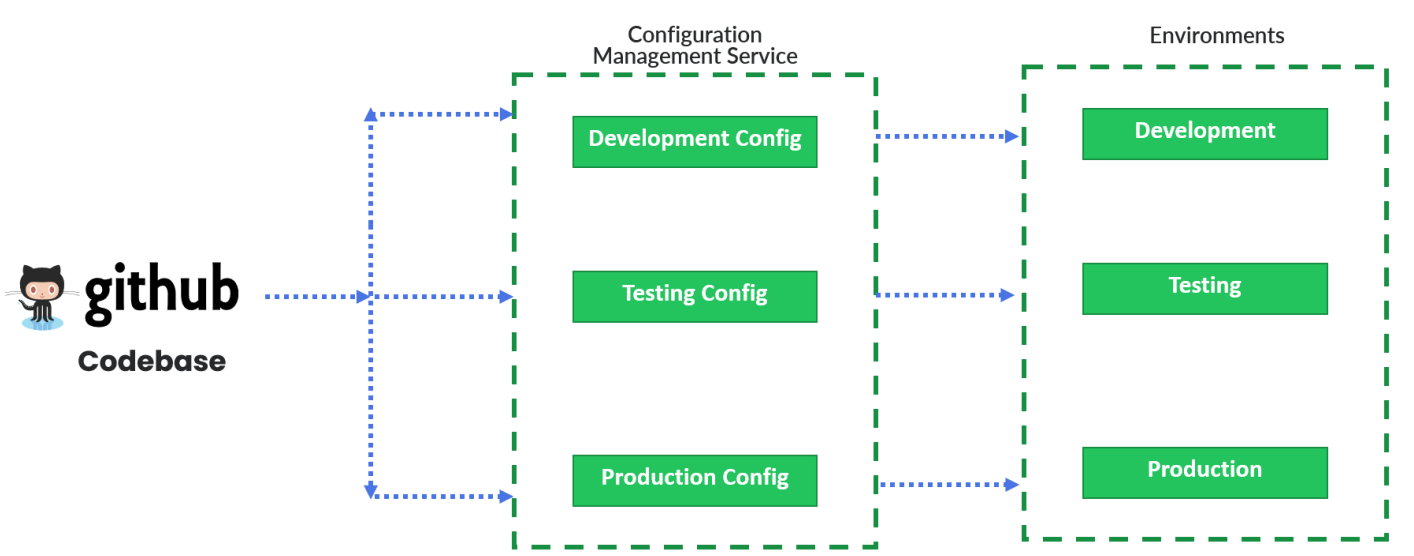




We can mange configuration of all microservices using above approach.

1. Maintain configuration management service
2. Maintain github as cental repo which holding configuration on env
3. After configuration will reflect for each microservice







It will manage all configuration for all microservices

1. Create spring initializer project with config-server dependencies
2. @EnableConfigServer on spring boot main app
3. Create folder inside resource like config and create your env based file like - accounts-dev.properties , accounts-prod.properties, accounts.properties.
4. In application.properties =
5. use native for profile.active because we are taking central repo as file system and use search-location as classpath:/config
6. if u want to get all config from file then paste your config folder inside C folder

and search-location as <file:///C://config>

1. Git Approach :

spring.profiles.active=git

spring.cloud.config.server.git.uri=https://github.com/ingleajay/microservices-config.git

spring.cloud.config.server.git.clone-on-start=true

spring.cloud.config.server.git.default-label=master

http:localhost:8071/cards/prod

In Account service if we want fetch all profile configuration from configserver services use below things :

1. In application.properties add :

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

1. Add some dependency in pom.xml:

<properties>  
 <java.version>17</java.version>  
 <spring-cloud.version>2021.0.3</spring-cloud.version>  
</properties>

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

<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>

1. Add models class like Properties :

@Getter  
@Setter  
public class Properties {  
 private String msg;  
 private String buildVersion;  
 private Map<String, String> mailDetails;  
 private List<String> activeBranches;  
  
 public Properties(String msg, String buildVersion, Map<String, String> mailDetails, List<String> activeBranches) {  
 this.msg = msg;  
 this.buildVersion = buildVersion;  
 this.mailDetails = mailDetails;  
 this.activeBranches = activeBranches;  
 }  
}

1. Add configuration like AccountConfiguration:

@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;  
  
}

1. Add Controller :

@Autowired  
AccountsServiceConfig accountsConfig;

@GetMapping("/account/properties")  
public String getPropertyDetails() throws JsonProcessingException {  
 ObjectWriter ow = new ObjectMapper().writer().withDefaultPrettyPrinter();  
 Properties properties = new Properties(accountsConfig.getMsg(), accountsConfig.getBuildVersion(),  
 accountsConfig.getMailDetails(), accountsConfig.getActiveBranches());  
 String jsonStr = ow.writeValueAsString(properties);  
 return jsonStr;  
}

URL : <http://localhost:8080/accounts/properties>

Put config server in docker and create image

We are facing problem previously in docker compose there is nothing with env like default and prod and dev but if we want our configuration will taken from github and provide configuration to each services through config server then you have to create sepreate docker compose file

version: "2.6"

services:

configserver:

image: ingleajay/configserver:latest

mem\_limit: 700m

ports:

- "8071:8071"

networks:

- ingleajay

accounts:

image: ingleajay/account:latest

mem\_limit: 700m

ports:

- "8080:8080"

networks:

- ingleajay

depends\_on:

- configserver

deploy:

restart\_policy:

condition: on-failure

delay: 5s

max\_attempts: 3

window: 120s

environment:

SPRING\_PROFILES\_ACTIVE: prod

SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/

loans:

image: ingleajay/loans:latest

mem\_limit: 700m

ports:

- "8090:8090"

networks:

- ingleajay

depends\_on:

- configserver

deploy:

restart\_policy:

condition: on-failure

delay: 5s

max\_attempts: 3

window: 120s

environment:

SPRING\_PROFILES\_ACTIVE: prod

SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/

cards:

image: ingleajay/cards:latest

mem\_limit: 700m

ports:

- "9000:9000"

networks:

- ingleajay

depends\_on:

- configserver

deploy:

restart\_policy:

condition: on-failure

delay: 5s

max\_attempts: 3

window: 120s

environment:

SPRING\_PROFILES\_ACTIVE: prod

SPRING\_CONFIG\_IMPORT: configserver:http://configserver:8071/

networks:

ingleajay:

@RefreshScope : this is for without restarting server it will take updated properties

encrypt.key=ajayingle

this is used when we want to keep our info is encryted so

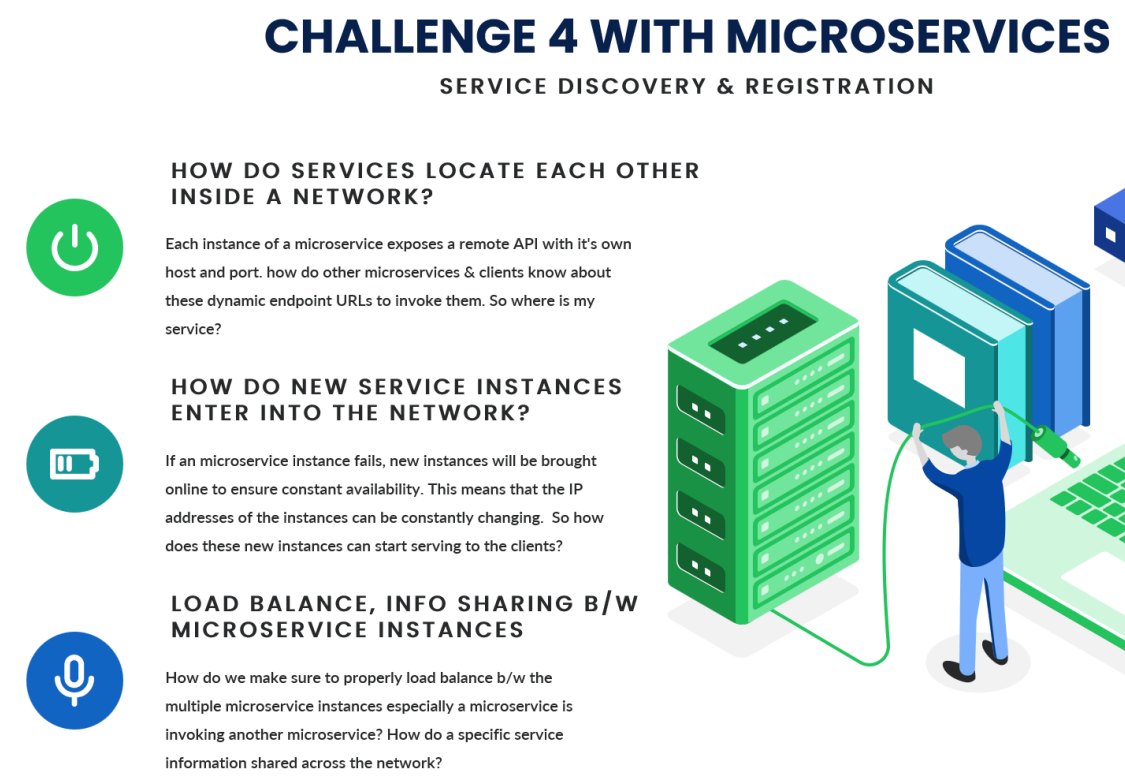
first you have to hit below url and get code

Post : as body pass text “welcome to loan prod” : localhost:8071/encrypt

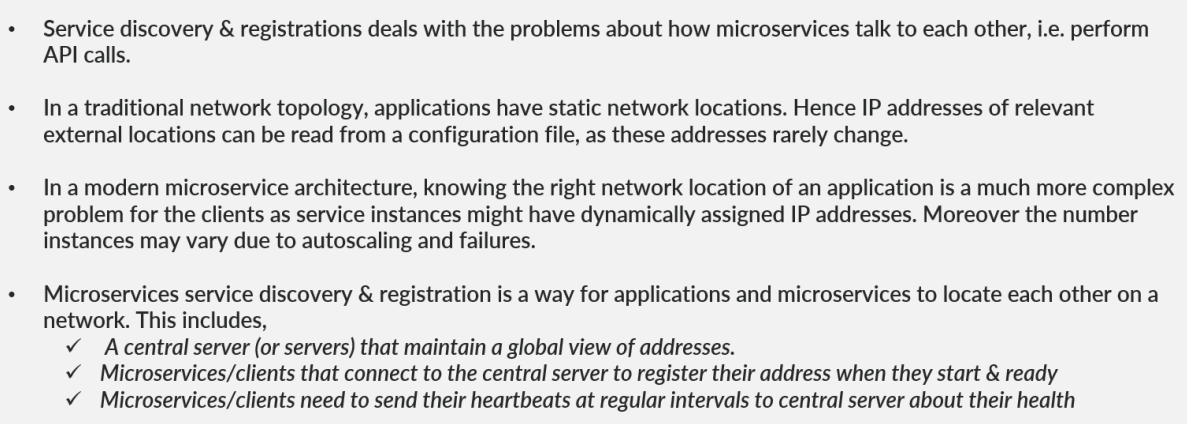
you will get code then paste that code into github after that you can check by localhost:8071/decrypt you will get your text

It is also reflected in your data which fetch by localhost:8090/loans/properties

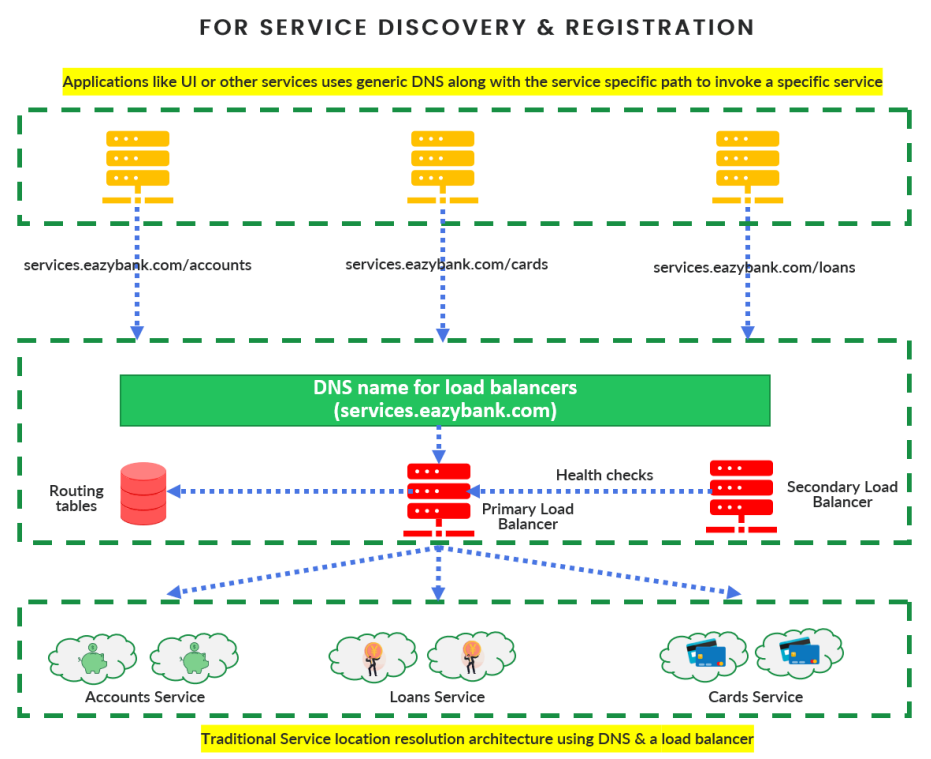
It will Next Challenge : how one microservice connects with other microservices like accounts wants to connect with loan then how it can be communicated



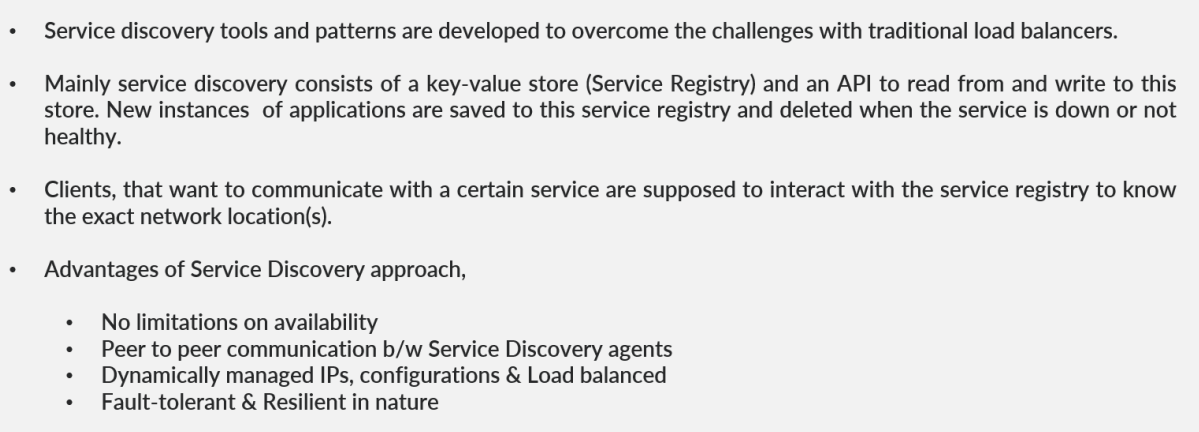
Use Server discovery :

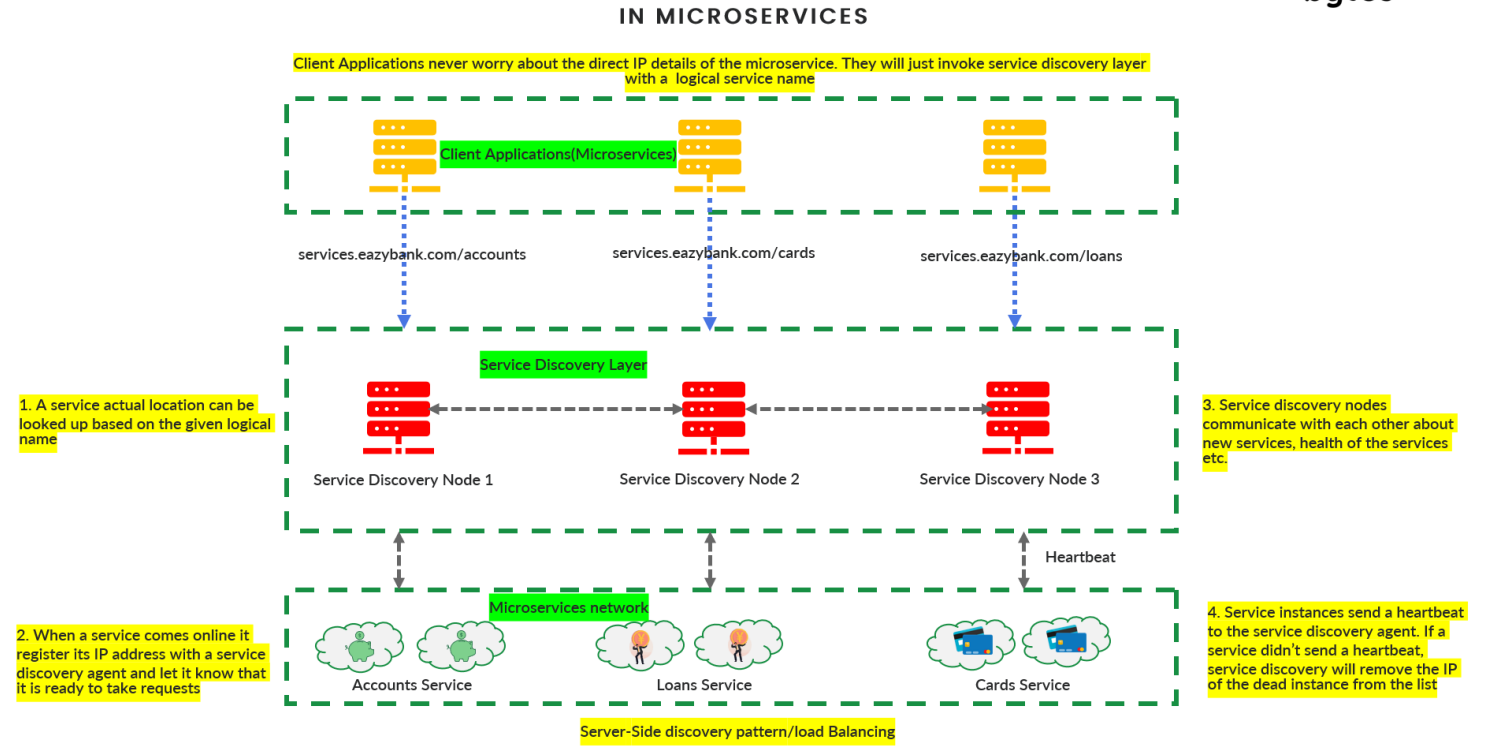


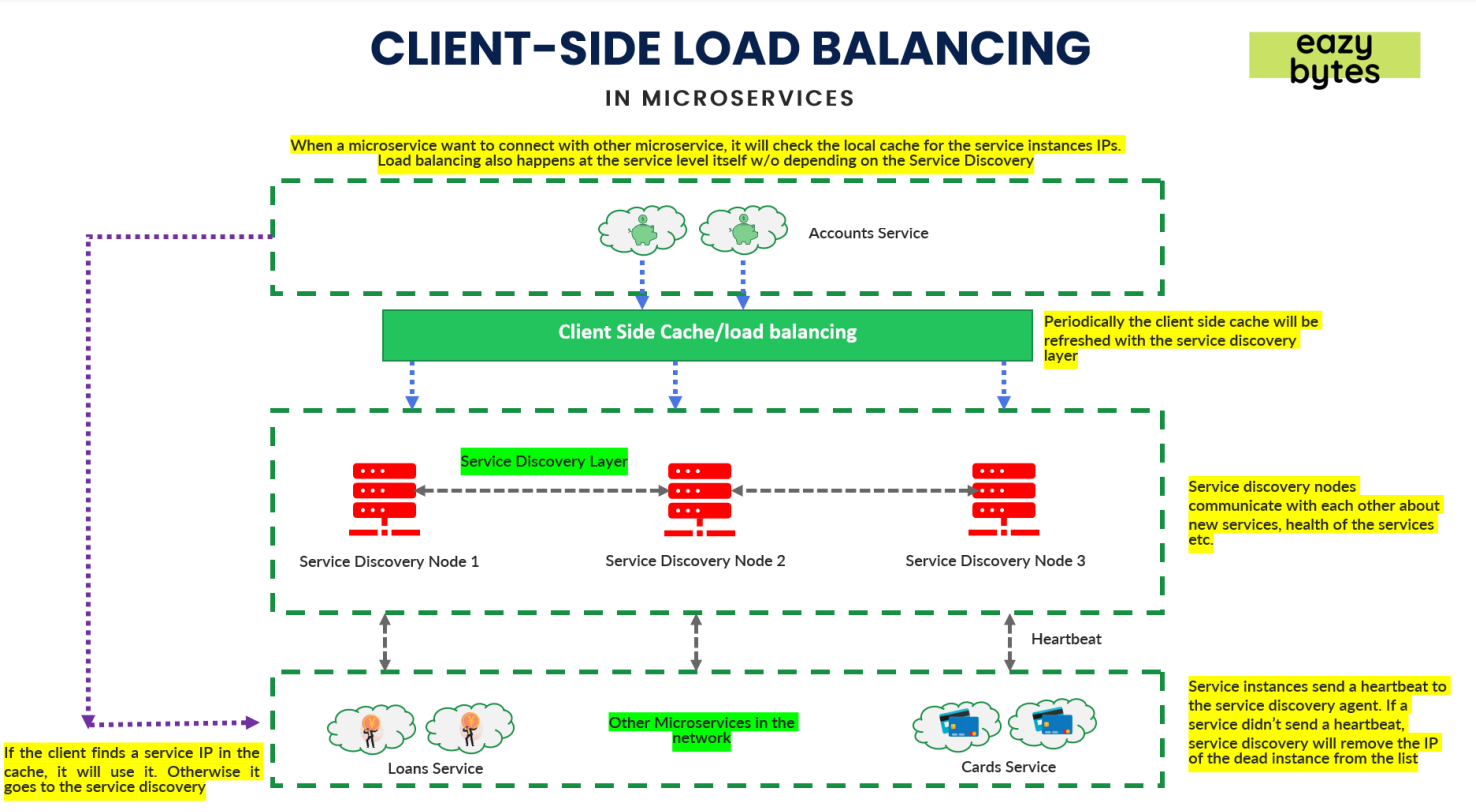
We maintain 100 microservice end url or points or ip and port in network so we will have one central server.

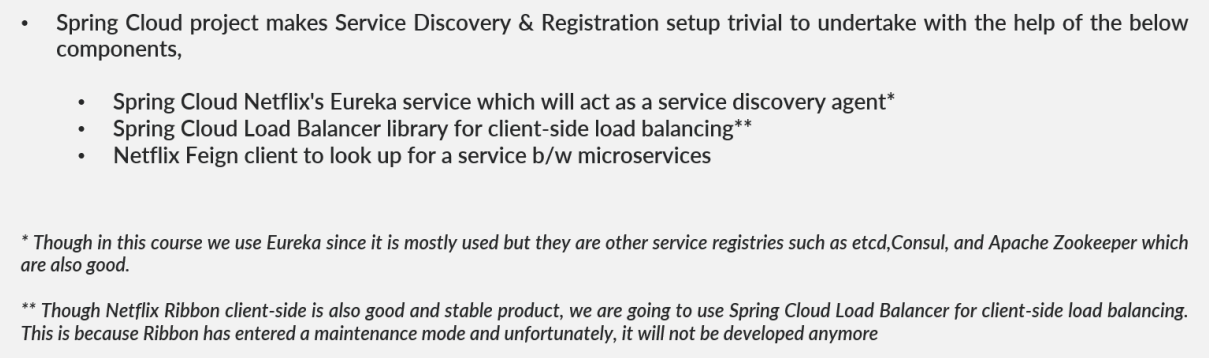


Load balanced is for change your starting name from your end point like localhost:8080/account = AccountService/account









Genrate euraka server by adding config-cloud, eureka server…

Add in app properties:

spring.application.name=eurekaserver

spring.config.import=optional:configserver:http://localhost:8071/

spring.cloud.loadbalancer.ribbon.enabled=false

Add property in config github :

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/

Every microservice will use eurka client to register with eureka server.

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

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

</dependency>

App prop :

management.endpoint.shutdown.enabled=true

endpoints.shutdown.enabled=true

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=Cards Microservice

info.app.description=Eazy Bank Cards Application

info.app.version=1.0.0

if you want to invoke card microservice into account microservice then u have to have dependency like openfeign of spring cloud starter

@EnableFeignClients use this is in account

localhost:8080/myCustomerDetails

you will get all cards and loans details

it is used to skip unit test of service

* mvn clean install -Dmaven.test.skip=true

We need to make microservice self resilience it means we have to make it in way that it handle difficult situation by there self.

